Erlend Stav

An Approach to Developing Extensible Application Composition Environments for End Users

Doctoral thesis
for the degree of doctor scientiarum

Trondheim, March 2006

Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics
and Electrical Engineering
Department of Computer and Information Science

NTNU
Norwegian University of Science and Technology

Doctoral thesis
for the degree of doctor scientiarum

Faculty of Information Technology, Mathematics
and Electrical Engineering
Department of Computer and Information Science

ISBN 82-471-7864-8 (printed version)
ISBN 82-471-7862-1 (electronic version)
ISSN 1503-8181

Doctoral theses at NTNU, 2006:58

Printed by NTNU-trykk

Abstract

Most software is now developed by professional programmers, and the millions of users of “shrink-wrapped” applications never meet the developers. The skills of professional programmers are obviously required in many software development projects. However, there are also reasons, ranging from practical and economical to philosophical and educational, why people without this background may want to develop their own applications or tailor applications to their own use.

The general purpose programming languages and development tools used by professional programmers are not suitable for people outside this profession. Instead, people outside this profession need more high-level tools that allow them to express solutions using domain and task specific terms. Despite the accumulated experience from such environments within the end-user programming community, creating a new application development environment for a domain or task remains a large development task, and can be too time-consuming and costly to be found worthwhile.

This thesis presents an approach to reducing the effort needed in developing extensible application composition environments for end users. We use the term “end user” in the same way as the end-user programming community, to denote a person who wants to develop an application or tailor an application to a specific use, without needing professional programming skills. The work in the thesis was initiated based on experience from development of proof-of-concept implementations of such environments in two European research projects. With this background, a set of user and developer roles are identified and organised into a value chain for development of such environments. Further, the approach combines the research areas of component-based software engineering and end-user programming to propose an overall architecture, component frameworks and tools suitable to support development of such environments. The validity of the results is established through partial prototyping of the frameworks and tools, combined with qualitative analyses of how these can be fully implemented and of their suitability for their intended purpose.

The main contributions of this thesis are:

- a value chain for development of application composition environments, with identification and description of tasks of each of the identified developer roles;
- an overall architecture for developing extensible application composition environments based on component frameworks;
- architecture of two component frameworks defining mechanisms and rules of behaviour for components ensuring extensibility at runtime and edit time;
- definition and description of a set of UML stereotypes for modelling domain frameworks based on the component frameworks, and a mapping to Java/JavaBeans allowing a code generator tool to produce part of the implementation;
- definition and description of tools which use the model of a domain framework as input and partially transform the work of creating composition environments and editors for domain objects from a programming task to a configuration task.

Acknowledgements

This is a doctoral thesis submitted to the Norwegian University of Science and Technology for partial fulfilment of the requirements to a dr.scient. degree. The work in this thesis has been done in the period 1998 – 2006, and has been supervised by Torbjørn Skramstad and Dag Svanæs. I would like to thank my supervisors for all their support, encouragement and thorough comments on my work during these years.

I am also very grateful to all others who have provided valuable input and feedback to this work. In particular I would like to thank Geir Skylstad, Jan Øyvind Aagedal, Jacqueline Floch, Hallvard Trætteberg, Arne Jørgen Berre and Frank Eliassen.

Finally, thanks to my family and my friends for their patience and support, and for being a continuous reminder that there are more important things to life than computer science research.

Trondheim, January 2006
Erlend Stav

Contents

Chapter 1 Introduction

1.1 Introduction

In a little more than half a century, electronic computers have developed and spread to be used in nearly every aspect of society. While the first computer users were often the same persons that built and programmed them, computers today are used for work, communication and entertainment purposes by a wide variety of people. Most software is now developed by professional programmers, and the millions of users of "shrink-wrapped" applications never meet the developers.

The skills of professional programmers are obviously required in many software development projects. However, there are also reasons why people without this background may want to develop applications or tailor applications to specific uses. The motivations can be:

- *Practical*. For many computer users, tailor-made software application could make their work or life easier. For some it is a case of automating repetitive work procedures, for others it can allow them to perform tasks they were not able to do earlier. Often the user already has the required domain or task knowledge. If a professional programmer is hired to make the software, this knowledge has first to be transferred to developer. If suitable tools were available to instead allow the user to create the software, misunderstandings could be avoided, and software better suited to tasks could be the result. It is also natural to think that the software would evolve as the user gets experience with its use.

- *Economical*. Professional programming help is expensive. There is also an increasing gap between the productivity of individual developers, and the amount of software used in products [Paternò 2003]. At the same time, the potential number of end-user programmers will significantly exceed the number of professional programmers in the foreseeable future [Scaffidi 2005]. In some organisations it is also acceptable to use a lot of internal/user time for a project, while hired external help is not acceptable at all. This may, e.g., be the case in institutions like schools and health care, or at home. While there are additional initial costs of creating software supporting end-user development, the possibilities it opens for adaptation to use context and organisational appropriation will lower the cost over time [Wulf 2004].

- *Philosophical*. There are also more philosophically grounded reasons for enabling end users to be software creators. [Fischer 1998] argues for a change of view on end users from passive consumers to designers. Software systems should be designed for evolution. Experience shows that they cannot be completely designed prior to use, and should evolve at the hands of end users.

- *Educational*. End-user programming is also interesting from an educational viewpoint. Papert's work on Logo was motivated by the idea that programming can help learning thinking skills [Papert 1980]. Children can also learn and explore a topic by programming a simulation of it using a suitable environment [Smith 2001].

1.2 Research Problem and Questions

The general purpose programming languages and development tools used by professional programmers are not suitable for people outside this profession. Instead these people need more high-level tools that allow them to express solutions using domain and task specific terms [Nardi 1993]. This is still a central part of the motivation in end-user development research, as expressed in the research agenda for the EUD-NET Network of Excellence project [Paternò 2003]:

> "*… one fundamental challenge for the coming years is to develop environments that allow people without particular background in programming to develop their own applications or modify existing applications, with the ultimate aim of empowering people to flexibly employ advanced information and communication technologies, especially within the upcoming ambient intelligence environments.*"

The end-user programming community has proposed a wide variety of tools and techniques, many of which have been successfully evaluated with users for limited domains and tasks. A disadvantage of making tools specific for a domain or task though is that it has limited generality, especially if the tool is not extensible. Despite the accumulated experience from such environments, creating a new application development environment for a domain or task remains a large development task, and can be too time-consuming and costly to be found worthwhile.

This situation motivates the research problem of this thesis:

- **How can we streamline and reduce the development effort required to build extensible application composition environments for end users?**

We use the term "end user" in the same way as the end-user programming community, to denote a person who wants to develop an application or tailor an application to a specific use, without needing professional programming skills.

The research problem places this work in the intersection between different fields of computer science. First, the motivation for the research problem is grounded in the research areas of end-user programming which belongs in the borderland between the fields of human computer interaction and programming languages. Secondly, the

question itself belongs within the field of software engineering, from which this work focuses on the research area of component-based software.

The work is further focused through a set of research questions that help approaching the problem:

- **Which user roles and developer roles are involved?** Finding and examining the user roles and developer roles gives a better understanding of which tasks are performed, how the division of labour can be improved, and where the greatest potentials for reducing the required development effort is.

- **What kind of reusable parts can be identified from existing end-user programming environments?** This work builds on the highly supported hypotheses in the software engineering field, that reuse is a key to making software development more cost efficient. Existing end-user programming environments described in the literature and available as software are assumed to be a valuable resource for finding parts that would be useful to have as reusable components when constructing new environments.

- **What kind of software architecture is suitable to build such environments?** Having a software architecture to reuse is useful because the work of creating the architecture does not need to be duplicated, but more importantly because it lays the foundation for reuse of components among systems.

- **How can reusable parts be integrated in new environments?** Experience in the field has shown that "as-is" reuse of existing components is often not possible [Bosch 1999]. The wiring-standards provided by the standard component models are important, but solve only part of the problem [Szyperski 1997].

- **What kind of tools will assist developers in building such environments?** In addition to applying reuse to reduce the work which needs to be done when creating an environment, tools can simplify the remaining work and make it more efficient. The tasks requiring most manual work holds the greatest potential effect if these can be supported by tools.

To answer these questions, our work has combined studies in the research areas of end-user programming and component-based software with results extracted from our own experience with two background case studies to propose a value chain, overall architecture, component frameworks and tools for development of extensible application composition environment. The validity of the results is established through partial prototyping of the frameworks and tools, combined with qualitative analyses of how these can be fully implemented and of their suitability for their intended purpose.

1.3 Computer Science Research Methods

Before describing the research method chosen to approach the research problem and research questions of this thesis, we will first visit some relevant work on scientific methods of the field.

In 1985 ACM appointed a task force on the core of computer science which resulted in a new intellectual framework for the discipline of computing. The main results of the work, which included both computer science and computer engineering in their scope, were presented in [Denning 1989]. Three major paradigms (or cultural styles) for how work in the discipline of computing is approached were identified, and the main sub-areas of the discipline were described. In the updated terminology of [Denning 2000], the paradigms or processes of the field are described as:

- *Theory*: "*Building conceptual frameworks and notations for understanding relationships among objects in a domain and the logical consequences of axioms and laws"*.

- *Experimentation (or abstraction)*: "*Exploring models of systems and architectures within given application domains and testing whether those models can predict new behaviors accurately"*.

- *Design*: "*Constructing computer systems that support work in given organizations or application domains"*.

In [Hartmanis 1994], the Turing award lecture of 1993, the nature of computer science was discussed. Although computer science is clearly not a physical science, the failure to apply the same paradigms to computer science as to the physical sciences has by some been regarded as an immaturity of computer science. Actually, computer science is a rapidly maturing science which should be regarded as a new species differing fundamentally from other sciences. In computer science, research paradigms and scientific methods are developed for a world of information and intellectual processes which are not directly governed by physical laws. In computer science, the relation between theory and experiments are different than in the physical sciences. Theoretical computer science does not challenge the basic model of computing, results are often of mathematical nature, and experiments are not used to challenge the results. Instead, in experimental work of computer science, system building is the defining characteristic, "*testing feasibility by building systems to do what has never been done before*". As a result, in experimental computer science, the "dramatic demo" can be seen to replace the "dramatic experiment" of the physical sciences.

The nature of the Experimental Computer Science and Engineering (ECSE) is also explored in [NRC 1994]. ECSE is defined as "*the building of, or the experimentation with or on, nontrivial hardware or software systems*". ECSE is a synthetic discipline, studying phenomena created by humans rather than those given by nature, and there is huge room for creativity and few direct physical constraints. The primary focus of ECSE is artifacts – software and/or hardware which are the subject of study, apparatus used to conduct the study, or both. Often a significant part of the intellectual effort of the experimental research is embodied in the artifact. Artifacts in ECSE can have one of the following three roles in the research:

- *Proof-of-performance*. The artifact shows that a certain performance can be achieved, or that it is in some other measurable way an improvement over previous implementations. The results are usually quantitative.

- *Proof-of-concept*. The existence of an artifact in this role proves that a concept is possible to realise, at least in one configuration. The artifact is usually too

complex to derive the behaviour of by only using logical reasoning or abstract argument.

- *Proof-of-existence*. An artifact in this role demonstrates through its existence a new phenomenon which is impossible or difficult to grasp only through documentation. A historical example of this is the computer mouse.

[Dahlbom 2002] discusses the need for a design-oriented science of artifacts. A traditional view dominating academic institutions is that science should be the discovery of scientific knowledge about a given reality. In such a view, scientist should refrain from invention to win scientific respectability. This view does not match current scientific practice, where even the natural sciences are more involved in invention than discovery. An artificial science should supplement the natural sciences and humanities as a third "faculty". [Dahlbom 2002] proposes the following slogan introduction for this idea: "*if natural science aims at explaining nature, the humanities at understanding human action, then artificial science is focused on the design of artifacts*".

While much of the research in computer science so far have not applied empirical approaches, empirical approaches are gaining recognition within the research community. In software engineering, examples of methods and guidelines for empirical research in software engineering are, e.g., [Kitchenham 2002] for quantitative methods and [Seaman 1999] for qualitative methods. In human computer interaction, methods such as observation and monitoring of users and usability engineering have found its way into the industrial product development as well as research praxis. [Preece 1994] describes and compares these methods and a set of other evaluation methods used in human computer interaction, also including collection of users' opinions, interpretive evaluation and predictive evaluation.

With this as a scientific background, the next subsection will describe the approach in this thesis in more detail and relate it to this background.

1.4 Thesis Approach

The work presented in this thesis mainly contributes to software engineering which is one of the sub-areas of computing defined in [Denning 2000]. In addition, the work is based on state-of-the-art from the topic of end-user programming which belongs in the borderland of sub-areas human computer interaction and programming languages.

The work in this thesis was initiated based on our experience from participation in two European research projects where environments for end user composition of software were developed. In each of these cases, a component-based architecture was developed for an extensible application composition environment. Proof-of-concept implementations of the environments were developed in both projects, and were tested by project participants without professional programming background [Head 1998], [TASC 2000]. The formulation of the research problem and research questions have grown out of the work on these cases, and the experience with these working solutions provided a good starting point for the architecture proposed in this thesis. In particular some of the mechanisms which proved to work well in the two cases have been reused in the proposed framework architecture.

The work is further founded on a study of the state-of-the-art in research literature and artifacts from end-user programming and software reuse with focus on component-based software. A set of approaches to end-user programming was studied and classified, with focus on a few selected example environments that have had great impact or are good examples of an approach. The purpose of this was to get a good overview of the kind of end-user programming techniques that should be supported by the frameworks proposed in this thesis. An assumption made here is that the state-of-the-art end-user programming techniques are working well, and no attempt has been made to enhance the techniques as such. Another assumption in this work is that software reuse techniques, and in particular component-based software techniques currently are the best available technological foundation for building extensible application composition environments.

The experience from the two cases and the studies of literature and artifacts was next applied to analyse which roles that could be identified in the development and use of extensible application composition environments. These roles were further organised to propose a value chain for development of such environments. The tasks involved for the most central roles were further analysed and described as use-cases which forms part of the requirements for the proposed frameworks and tools. With the use-cases as background, existing environments were analysed to see how these environments solved the tasks, and to which extent and how these solutions could be reused in the development of new composition environments. This analysis also contributes with further requirements on the frameworks.

With this background, the approach to developing extensible application composition environments for end users was developed. An overall architecture including two component frameworks were defined; a foundation framework used at both runtime and edit time, and an editing framework defining rules ensuring collaboration between different editors. Based on these frameworks, a set of tools and methods were proposed which will further simplify and make the task of creating composition environments more cost efficient.

Partial proof-of-concept prototypes were made for the proposed frameworks and tools. Important parts of the proposed frameworks were derived from the two case studies in which complete working pilots were implemented, and the prototyping was focused on mechanisms and tools that were not derived from the environments of the case studies. The validity of our results were further considered by analysing how the proposed frameworks and tools can be fully implemented, whether they can be used for their intended purpose, and their suitability for this purpose. Further, qualitative analysis were used to discuss how this work contributes to a clearer division of labour and reduction in the development effort required to design, code and move from design to code in the development of composition environments.

Compared to the paradigms or processes defined by [Denning 2000], the work presented here belongs partly in the design paradigm and partly in experimentation. The case studies had strong elements of design with a focus on creating good composition environments for their user groups. On the other hand, their construction was far from trivial, and the implementations can clearly be regarded as proof-of-concept for an experimental work verifying the suitability of the selected architectures for building extensible composition environments. Similarly, the work on the architecture,

frameworks and tools proposed belongs partly in the design paradigm and partly in the experimentation paradigm.

The work presented in this thesis mainly has a technical focus, but part of its motivation is that a full realisation of the ideas would enable more people to develop their own software or tailor software for their own use. The choice of topic is thus value oriented and engaged rather than based on objective detachment, which according to [Dahlbom 2002] is one of the characteristics of artificial science.

1.4.1 Limitations and Timing

This thesis proposes tools and methods for a broad set of activities in the development of extensible application composition environments. By selecting this approach, it has not been possible to go into depth on each tool and method, but in-depth experience with important parts of the proposed solutions was gained in the two case studies.

As mentioned above, partial proof-of-concept prototypes were made for the proposed frameworks and tools, with a focus on parts that were not derived from the environments from the two case studies. Complete implementation of the proposed frameworks and tools was not possible – this would have required substantially more time and resources than have been available within this thesis work.

Another limitation with the work reported in this thesis is that it does not include any empirical studies on actual use of the proposed frameworks and tools. A full empirical study of the framework and tools would also require a more complete implementation and a lot more work.

The main part of the work on this thesis was performed in the years 1998 to 2002. More recent related work has not been used as part of the foundation for this thesis, but a discussion of the most important of recent related work, including Model Driven Architecture (MDA), Eclipse, Microsoft's Software Factories, and End-User Development research is presented in Chapter 9.

1.5 Claimed Contributions

With suitable domain specific application composition environments, users without professional programmer training are able to create new software applications. The main contribution from this thesis is an approach to building such environments. The approach combines the research areas of component-based software engineering and end-user programming to propose an overall architecture, component frameworks and tools suitable to support development of such environments. The approach includes:

- a value chain for development of application composition environments, with identification and description of tasks of each of the identified developer roles;
- an overall architecture for developing extensible application composition environments based on component frameworks;
- architecture of two component frameworks defining mechanisms and rules of behaviour for components ensuring extensibility at runtime and edit time;

- definition of a set of UML stereotypes for modelling domain frameworks based on the component frameworks, and a mapping to Java/JavaBeans allowing a code generator tool to produce part of the implementation;

- definition of tools which use the model of a domain framework as input and partially transform the work of creating composition environments and editors for domain objects from a programming task to a configuration task.

1.6 Overview of the Thesis

Following this introductory chapter, Chapter 2 gives a presentation of end-user programming. It covers both theoretical background in the field, and examples of available software. The reason for including this chapter is to give a brief overview of the state-of-the-art in areas of end-user programming including end user tailoring. Further it is used as a resource for further analyses to find potentially reusable elements that occur in such software.

Chapter 3 presents a brief description of state-of-the-art in component-based software development technology and software reuse. The purpose is to present what would be the current technology of choice for implementing extensible application composition environments. The chapter also covers object orientation, software architecture and patterns, and component frameworks.

The cases studies described in Chapter 4 are projects we have been involved in where task-specific development environments where developed. With these cases we supplement the previous chapter based on in-depth first-hand experiences with architecture, frameworks and components from their development. The chapter also describes an additional example which is used to give concrete illustrations of how the proposed frameworks and tools can be applied.

Chapter 5 describes a set of user and developer roles, and organises these in a value chain for development of domain- and task-specific application composition environment. The tasks of the most central roles are then analysed and described using use-cases.

Chapter 6 analyses the tools and techniques use in some of the end-user programming tools and the case studies. Reusable parts from these are identified, and is organised by their areas of concern.

In Chapter 7 an overall architecture for development of extensible application composition environments is proposed. Further, two component frameworks are proposed: a Foundation Framework defining mechanisms required both at runtime and edit time, and an Editing Framework defining how a set of editor components can be combined in an environment. Each framework is first presented with a platform independent description, and then mapped to Java/JavaBeans as an example of a realisation platform.

Chapter 8 proposes a set of tools which can assist the different developer roles, with primary focus on the domain environment developer. A section in this chapter also describes the degree to which the proposed frameworks and tools have been prototyped.

Chapter 9 discusses the various aspects of the proposed approach and relations to some recent research and development. The research method and validity of the results is also discussed, before conclusions are drawn on the research questions and research problem. The chapter and thesis closes with some suggestions for future research.

1.7 Terminology

The following list defines a set of frequently used terms within this thesis.

Term	**Description**
(Application) Composition Environment (*for end users*)	An environment consisting of a set of software tools enabling people without professional programming background to compose software applications within a limited domain or for a limited set of tasks
Architecture	"*The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution*." [IEEE 2000]
Component	"*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties*." [Szyperski 1997] One kind of *reusable part*.
Component Framework	"*A component framework is a dedicated and focused architecture, usually around a few key mechanisms, and a fixed set of policies for mechanisms at the component level*." [Szyperski 1997]
End-user programming	The activity of creating, adding to or modifying the functionality of software, in a way that can be performed without professional programming education and training.
Extensible	Used to describe a software system which is open to have its initial functionality extended, typically by adding a new component to the system.
Framework	"*In software development, a Framework is a defined support structure in which another software project can be organized and developed. Typically, a framework may include support programs, code libraries and a scripting language amongst other software to help develop and glue together the different components of your project.*" From http://en.wikipedia.org/ (accessed 2005.06.05)
Model	Simplified description of a system. In this work, usually expressed in UML. (exceptions layer model and architecture reference model)

Pattern	"*Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice*". [Alexander 1977] (page x)
Reusable part	Broad term for something that is reusable, including a concept, a mechanism, a piece of source code, a binary *component* etc.
Role	When the term is used the context of users and developers of software, a role denotes a function performed by a person within the overall development and use of a software system. A single person can in some cases play multiple roles, and a role can also be filled by multiple persons.
Software reuse	"*Software reuse is the process of creating software systems from existing software assets rather than building software systems from scratch.*" [Krueger 1992]
Tool	Software utility that helps a user or developer to perform some task or part of a task.

Chapter 2 End-User Programming

When “grand visions” of the future are presented, computer users often verbally instruct their personal “software agent” of what they want to do. Although there are advances in technologies necessary to realise these visions, e.g., speech recognition and agent technology, end-user programming remains a more realistic approach to some of the scenarios these visions present [Nardi 1993]. End-user programming approaches seek to enable users without professional programming background to create, modify or integrate software applications to suit their own needs, usually by using high-level tools and visual formalisms that allow the users to express solutions using domain and task specific terms.

This chapter gives an overview of the research area of end-user programming, and a brief overview of end user tailoring research. Motivations for end-user programming were described in the introduction chapter. In the first section of this chapter the question “what is end-user programming?” is addressed. The different user and developer roles involved are described next, followed by a section on the approaches to end-user programming, and a further look at some environments available commercially or in the research community. A section on end user tailoring is presented, before the short summary closes the chapter. The relation of our work to recent end-user development research is discussed later, in Section 9.2.4.

2.1 What Is End-User Programming?

A question that may be reasonable to ask is: when is something termed end-user programming, and when is it just adjustment of pre-selected options? One might argue that to classify something as programming, at least some new behaviour or structure should be defined. There will always be cases that are difficult to classify. E.g. how about options that allow selection among a set of predefined behaviours, or selection of multiple such behaviour options that may depend on each other? [Nardi 1993] addresses this question: *“The objective of programming is to create an application that serves some function for the user. From the end user’s point of view, the particular behaviour involved is not important, so long as application development is easy and relative rapid. In this respect, we can include automatic programming system, programming by example systems, and form-filling dialogs in which applications can be customized.”* The continuum of programmability that this suggests is presented in Figure 1 borrowed from [Nardi 1993]. The figure shows a set of technologies enabling end-user

programming, and arranges these along an axis of increasing expressiveness ranging from parameter settings, via behaviour concatenation and creation, to traditional programming. The other axis shows the degree to which the technique supports interactive construction. Although the examples in the figure are old, the axis and categories of expressiveness used remain valid.

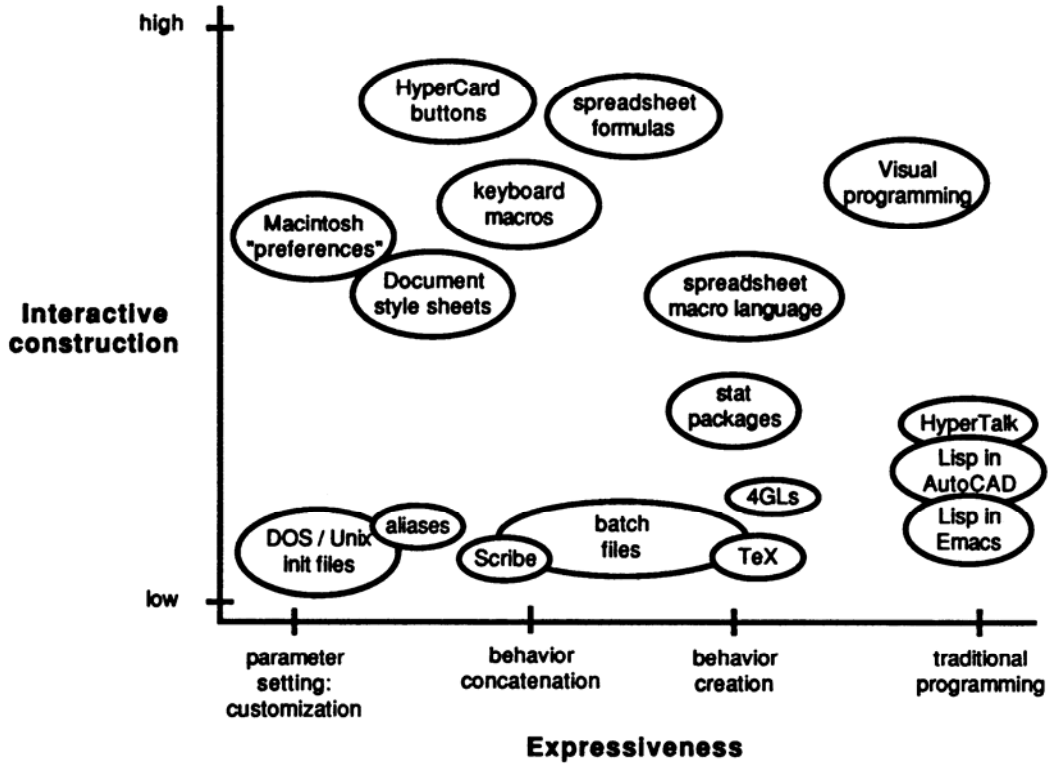

Figure 1 - Expressiveness and interactive constructs in end-user programming systems [Nardi 1993]

The term “end-user programming” is understood somewhat more broadly than its literal meaning: “programming by people who are not professional programmers”. Thus it includes making software for other persons, such as colleagues and pupils. Another term used has been “non-programmers”, but some in the community think this is a too negative term [Nardi 1993], as it indicates they are not able to create programs.

So what is created during end-user programming? One way to group the wide range of possible results is to look at how much of an application the user creates:

- *New applications*. In some cases whole new applications are built more or less from scratch. Examples here are tailor made tools, e.g., for pupils or disabled users [Svanæs 1993]. Other examples are the simulations created by tools like StageCast Creator [Smith 2001] and Agentsheets [Repenning 1993], and interactive presentations like the one produced with Macromedia Director.

- *Modifications and additions to existing applications*. This can take the form of modifications that tailor some of the behaviour of an application to suit the

user's preferences, for example, in the form of training of "agents". Or it can be automation of repetitive tasks the user frequently performs.

- *Integration between applications*. This is similar to the previous point, but includes multiple applications. With end-user programming users may be able to integrate applications in ways that provides new functionality, or that makes their work quicker or easier.

To sum this up, the term "end-user programming" will in this thesis be used to mean the activity of creating, adding to or modifying the functionality of software, in a way that can be performed without professional programming education and training.

2.2 User and Developer Roles

Although there is some variation among approaches, traditionally the following developer and user roles can be identified in the development and use of end-user programming environments:

- *End user*. The user of the finished software application.
- *Application developer*. The developer who uses an end-user programming environment to create an application for the end user.
- *Environment developer*. The developer who creates the end-user programming environment used by the application developer.

As the literal meaning of the term suggests, in end-user programming the end user and application developer role are often played by the same person.

In the studies of [Nardi 1993] there were sometimes local developers or programmers who helped the end users in the development by developing macros or teaching them to do something new. In some of the organisations of the studies this role was formalised, and these *gardeners* where given resources and recognition for the work. The gardener role can be seen as taking over the most advanced development work from the application developer, leaving the application developer role with simpler development tasks. [Repenning 1993] also describes a richer set of user and developer roles which emerged in the Agentsheets studies, and these are described in Section 2.3.5.

Increasingly, end-user programming environments are extensible with plug-ins that add new functionality that the application developer can utilise. Hypercard's XCMDs was an early example in this area. This extensibility adds the role of a *plug-in developer.*

2.3 End-User Programming Approaches

This section gives an overview of the main approaches to end-user programming. Specification of behaviour is the main characteristic that separates end-user programming from other software use, and thus the approaches are categorised by how behaviour is specified. Within the description of each approach examples are given of end-user programming environments using the approach. The selection of environments

presented is not extensive, but instead focuses on a few examples that have had great impact or are good examples of the approach. Some environments combine elements from different approaches, and these are included under their primary approach. The approaches described are:

- general purpose programming languages;
- scripting;
- macro recording;
- programming by example;
- grid-based environments;
- visual programming;
- other approaches and research.

2.3.1 General Purpose Programming Languages

General purpose programming languages have usually been developed with the professional programmer in mind. As the term suggests, these languages are intended to be suitable for any, or at least a broad set of purposes. Some early and optimistic views on end-user programming seemed to think that it was just a matter of making a visual programming language. [Nardi 1993] argues that there are other issues that are even more important. End users readily learn formal systems, when motivated to, but the language should be suitable for the task. Thus a general purpose programming language, visual or not, is not suitable for end users.

One of the early attempts in this category is Papert's work on Logo [Papert 1980]. Logo is a powerful language, but both the language and its interpretive environment were designed to help children learn to program. From the environment a "turtle" could be instructed to move around and using a pen to draw along the path it moved. These commands could be executed with immediate feedback. The children could gradually advance to define procedures as sequences of instructions. The idea of the Logo work was both to give children a gentle way to learn programming, and also that learning to program would help to develop abstract thinking skills.

Early versions of the Smalltalk programming language was initially also developed with children and end user programming in mind. The language was developed as part of the Dynabook research at the Learning Research Group of Xerox PARC in the 1970s. The Dynabook was envisioned to be personal computer the size of a notebook, giving the user a dynamic media allowing them to create animations and music [Kay 1977]. In addition to the language itself, the Smalltalk environment included a mouse-based user interface with multiple overlapping windows which later greatly affected the development of modern windowing systems. The Smalltalk language should allow the users to become "writers" in the new dynamic medium, and develop their own programs. Interim versions of the Dynabook (using custom built Xerox hardware) were tested by children, and [Kay 1977] describes some successes stories were children were

able to create a painting system and a music score capture system. In retrospect, [Kay 1993] still sees these successes as real, but not as general as hoped, and attributes part of the success to the fact that, "*for any given pursuit, a particular 5% of the population will jump into it naturally, while the 80% or so who can learn it in time do not find it at all natural*". In the later development of Smalltalk at Xerox, from Smalltalk-76 and to the release of Smalltalk-80, the focus for development changed more to supporting professional programmers. Regarding extensibility, Smalltalk of course allows the use of third-party Smalltalk libraries. Also, the language has a concept of primitive methods which is used to implement the most platform-near part of the standard libraries, but can also be used to develop extensions beyond what is achievable directly from within the language.

One of the most widespread environments in this category is Microsoft's Visual Basic[1]. This environment is built around the BASIC programming language. BASIC was from the start intended to be simple to use for beginners in programming, but as a general purpose programming language share the problems of these for end users. The main reason for the success of Visual Basic is probably its user interface builder which enables visual composition of the user interface. Visual Basic was also an early success story for components and established a component market, as new object types (both visual and non-visual) could be added to the environments in the form of binary components developed by third-parties. Although the user interface builder makes it easy to create the static parts of the user interface, the organisation of the rest of the application must be managed by the user, with the BASIC language as the main tool.

A more recent attempt to create a general purpose language for children is ToonTalk[2] [Kahn 2001]. ToonTalk is built on the idea of animated programming. A ToonTalk program is a collection of autonomous processes that communicate asynchronously. Concurrent objects are represented as houses, guarded clauses as robots, data structures as boxes, and send/receive by birds and nests etc. Robots are trained in new behaviour by the programmer by showing him what to do. The concrete demonstrations are generalised by removing detail – this is done using a "Vacuum cleaner" tool. A clause accepting the number 1 as input can, e.g., be generalised to take any number by using the "Vacuum cleaner" tool on it, or by using the tool twice input of any type will be accepted. ToonTalk also have an extension mechanism, allowing calls extension libraries created as Windows DLLs. From the language, this appears as special kind of bird that can fly "outside" the system.

2.3.2 Scripting

Scripting possibilities are included in a wide variety of software. Scripting language are supposed to be easier to learn and use than a general purpose programming language. Thus these languages tend to be less rigid, e.g., by not requiring variable declarations, having a more "relaxed" syntax, and an interpreted environment. The languages can also be task or domain specific, making it easier for the users to understand and express the behaviour they want.

[1] http://msdn.microsoft.com/vbasic/ (accessed 22.Sept 2005)
[2] http://www.toontalk.com/ (accessed 22.Sept 2005)

Being simpler languages, scripting languages usually have some limitations compared to general purpose languages. As scripts are expected to be simpler and smaller than full-scale programs, abstraction mechanisms for organising large programs may not be included. Also low-level features like explicit memory allocation are not common. While this is limiting some of the freedom of the script programmer, it also reduces the chance that the programmer may do anything seriously wrong that will lock up the system, or even corrupts other parts of the system.

While adapting the scripting language to each domain and task is preferable from some viewpoints, it may cause a problem if the user needs to use many different tools, each with its own language. One of the more widespread scripting languages, Microsoft Visual Basic for Applications[3] (VBA), has taken a different approach. The same language is used in many of Microsoft's applications, such as Word, Excel and PowerPoint. Another example of a domain-independent, system wide scripting language is AppleScript. This language supports scripting across applications on the Macintosh platform.

HyperCard[4] was an early example of a system allowing creation of new applications from scratch through a combination of visual editing of a card stack and using scripts for behaviour. As long as the applications fitted well with the card stack metaphor, this was a productive environment for creating simple software applications and prototypes. The scripting language of HyperCard, HyperTalk, used an English-like style, with commands like "Go to the next card". Although friendly to read, the language is far from perfect, and, e.g., has problems with lack of consistency [Thimbleby 1992]. [Nardi 1993] claims that the compromise between a task specific language and general programming language found in HyperTalk is a bad one, not being close enough to end users, and not powerful enough as a general language. HyperCard supports a limited kind of third-party extension to the system in the form of XCMDs which add new functions that can be called from HyperTalk.

With the growth of the Internet, scripting languages has also come in widespread use in web pages. Scripts in languages like JavaScript and VBScript are used in the web pages and interpreted by the client's browser, while on the server side scripts in languages like PHP and Perl are used to create dynamic content based on user requests like forms, and accessing server databases.

Syntax problems have often been suggested as a problem with text-based languages. Based on the studies of how users manage the formula language of spreadsheets, [Nardi 1993] concludes that proper syntax checking appears to be sufficient to deal with this. The formula languages of spreadsheets are simple compared to most scripting languages though, and it is not clear if this observation is valid more generally.

2.3.3 Macro Recording

The idea of macro recording is to record a sequence of user actions that can be useful to repeat. The user usually has to specify when recording starts and ends, and all actions in between are considered part of the macro.

[3] http://msdn.microsoft.com/vba/ (accessed 22.Sept 2005)
[4] http://en.wikipedia.org/wiki/HyperCard (accessed 22.Sept 2005)

There are two main problems with macros:

- *Macros are usually too specific*. The actions of the user usually refer to objects that should be modified in some way. Simple macro recorders remember exactly these objects, and performing the macros again would affect the same object. Often what the user want is to repeat the same actions applied to different objects. Programming by example approaches (presented in the next subsection) attempts to solve this problem.

- *Macros are recorded as scripts*. Usually macros are recorded as scripts, and thus to modify or even look at what the macro does without performing it, the user has to understand the scripting language [Repenning 2001]. Macros in Word and Excel are, for instance, recorded as Visual Basic for Applications scripts.

Some systems attempt to remove the need for the user to explicitly start and stop the recording. In the Dynamic Macro approach [Masui 2001], whenever a repeat key is pressed, the system looks for repetitive operations in the history of recent user's operations. If such a sequence is found, a macro is defined and executed. Additional repeat key presses will replay the macro.

2.3.4 Programming by Example

In programming by example (PBE) the user demonstrates one or more examples of what the program should do, and the program is derived from the user's demonstration. PBE is similar to macro recording, but differs in that the approaches attempt to solve the problem of macros being too specific. Two good sources containing papers by leading researchers in the field are [Cypher 1993] and [Lieberman 2001a].

Programming by example, or programming by demonstration, as it is also called, has been applied to a wide set of problems. These vary in complexity (for the user) from small adaptations of applications, to the creation of new applications from scratch. [Masui 2001] describes the use of PBE for adopting predictions used for efficient composition of text on small devices. In [Lieberman 2001b] agents are trained to recognise text, gradually building a grammar for the recognition by example. [Masuishi 2001] describes how reports can be built using PBE, with the user specifying the format using copy and paste of data from an example row, and specifying iteration by selecting a pattern and demonstrating using another row of data. ToonTalk, which has already been described in the general purpose language section, uses PBE to specify the behaviour, and create an application from scratch.

Another interesting example in is the Gamut system [McDaniel 1999] [McDaniel 2001]. The main claim in [McDaniel 1999] is that PBE can be used to build whole applications. Gamut is a domain specific system that is intended for making games, simulations and educational software. It is based on a deck-of-cards metaphor. The deck-of-cards are used both for making multiple screens by showing only the top card and for random functions through shuffling of the deck. Cards allow users to define "records" by adding objects to the cards. In addition to the objects that are displayed at the cards at runtime, the cards also has an off-screen area that can contain guide objects. A card containing a drawing of a dice can, e.g., have the dice's numerical value stored off-screen. The system also allows on-screen guide object that are invisible at runtime.

An example of these is paths of arrows that visual objects can follow. Figure 2 shows the Gamut environment being used to create a Pac-man game, where the paths are used to show where the Pac-man and monsters are allowed to move. The Gamut system has two buttons that is used when behaviour should be recorded, called “Do something” and “Stop that”, shown at the bottom right of the figure. As the names suggest these are used for recording new behaviour and for modifying behaviour that is not as intended. During demonstration of behaviour the user may specify hints by highlighting objects that are relevant. The mentioned paths of arrows are examples of objects that can be highlighted in such situations. Temporal ghosts are used for referring to previous states of modified objects during demonstration.

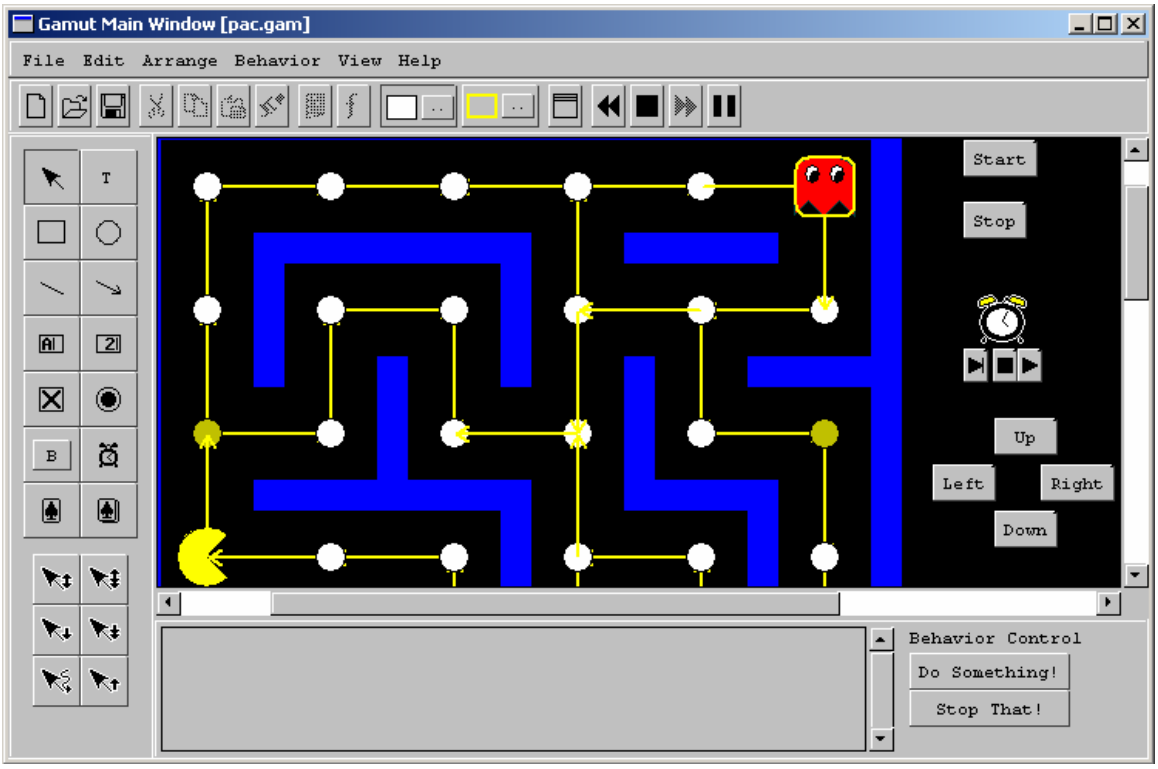

Figure 2 – Gamut environment with Pac-man game example

As the system has to derive a generalisation of the problem from one or more examples, AI techniques are often applied in this area, and much of the research is the area are primarily on AI problems. From the available description, Gamut does not appear to have any mechanism for creating third-party extensions to the system.

2.3.5 Grid-Based Environments

Stagecast Creator[5] (earlier KidSim) [Smith 1994] [Smith 2001] is a grid-based end-user programming environment. The user creates an application by placing *characters* on a page that is divided into a grid of equally sized cells. Each character has one or more appearances that can be edited, and a set of rules that decides the behaviour. The rules are organised as an ordered list.

[5] http://www.stagecast.com/ (accessed 22. Sept 2005)

The rules of the characters are created “by example”. The definition of a rule usually starts by selecting the character and optionally a set of neighbour cells that is part of the context for the example. The contents of these cells when the rule is created are part of the precondition for the rule to trigger. The user can then demonstrate the effects of the rule by modifying the situation. This typically involves changing the content of one or more of the involved cells, changing the appearance of one or more characters, or setting some other variable. The effects can later be viewed and edited further as a list of actions. Figure 3 shows how a rule for moving an actor to the right on the grid is created in Stagecast Creator. The left part shows the grid and the selection of cells used when creating the rule. The right part shows the rule maker dialog, with the rule illustrated through the situation to match (to the left of the arrow) and the change to be applied (to the right of the arrow).

The execution of the application is divided into time steps. For each time step the rules of each actor are examined, and the first rule of each actor which matches the preconditions (if any such is found) is executed.

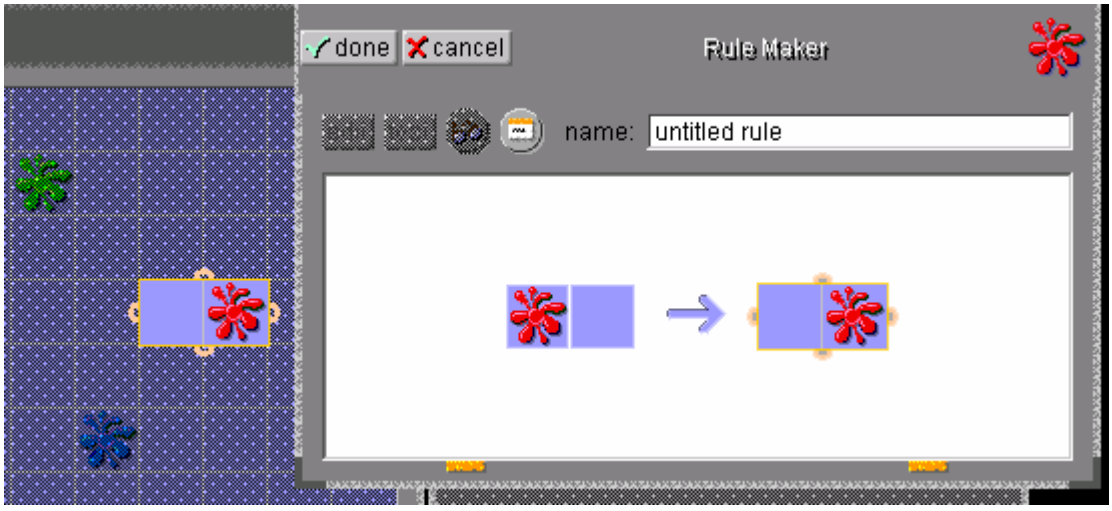

Figure 3 - Creating a rule in Stagecast Creator

The basic principles of Agentsheets[6] [Repenning 1993] are very similar to Stagecast. In the Agentsheets construction paradigm a number of autonomous and communication agents are organised in grid-based containers called agentsheets. [Repenning 1993] defines four different user and developer roles around the Agentsheets construction paradigm, where the roles work on different layers of the system:

- *End-users* work in the application artifact layer, and customise application scenarios by simple operations such as adding and removing agents and changing parameters, and play what-if games by executing the scenarios.

- *Scenario designers* shape original content of the application artifact layer by composing scenarios from domain-oriented agents and agentsheets.

[6] http://agentsheets.com/ (accessed 22. Sept 2005)

- *Spatio-temporal metaphor designers* design notations and agents for a problem domain by creating subclasses of agents and agentsheets, and thereby form the domain oriented applications layer.

- *Substrate designers* create the substrate with support for tasks common to a broad set of application domains, and thereby build the problem solving-oriented construction paradigm layer.

The Agentsheets tool is slightly less visually oriented than Stagecase. Instead of selecting an area of neighbour cells, conditions like “See a” and “Next to” are used to check on the content of the *agent’s* neighbour cells. Instead of demonstrating the effects of the rule, the user inserts a set of actions like “move”, “new” or “erase”. Figure 4 (from [Repenning 1993]) shows an example application from a voice dialog design environment built in Agentsheets. The visual language in this voice dialog environment support prototyping of the interaction between people and phone-based information services.

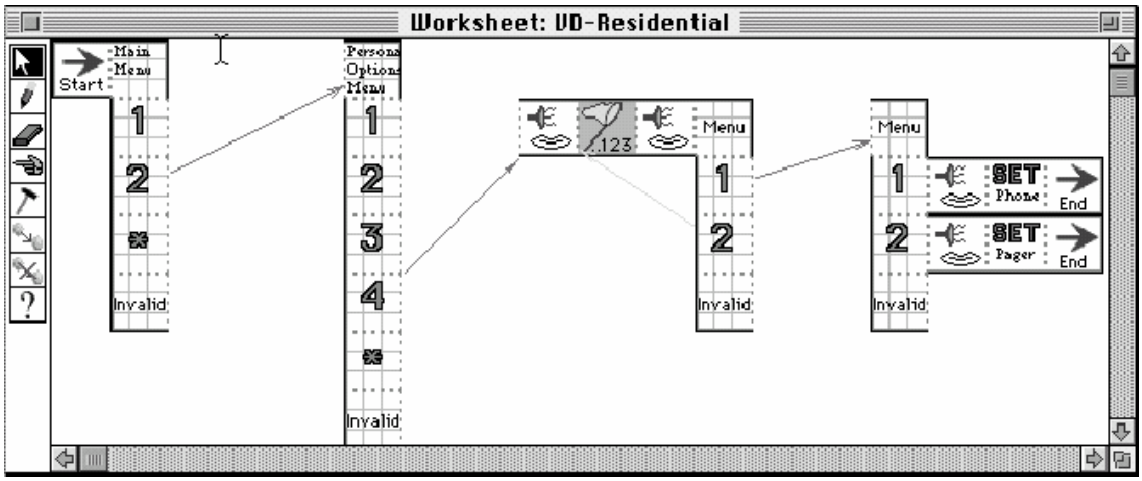

Figure 4 - Voice dialog example from Agentsheets

Neither Stagecast creator nor Agentsheets seem to have any generally available mechanisms for creating and using third-party extensions, although for Stagecast information about a Java-based SDK for extending the environment is available by contacting the company.

One of the reasons these tools seem to be successful is that the grid-based model is very simple to relate to and to understand. The application is nothing more or less than the grid and the characters/agents (called actors from now on) contained in the cells. There is not any hidden, more complex “reality” that causes unexpected and unexplainable behaviour, although in more complex situations the application can show emergent behaviour not expected by the user as a result of interaction between the rules of multiple actors.

Another reason for their success is probably an easily understandable execution model. Somewhat simplified, the execution is divided in steps, and each actor uses the first rule in its list that matches the precondition (although it is also possible to include more event-based tests like checking for a pressed key, or doing something, e.g., each second).

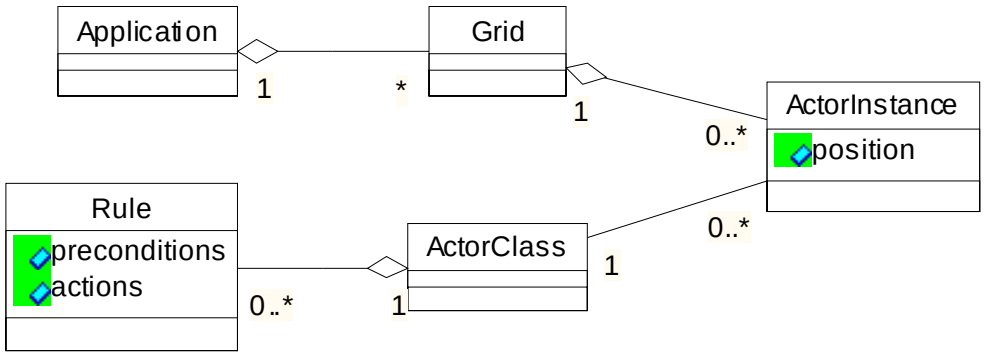

Figure 5 - UML class diagram for the main object types in grid-based environments

A simple UML class diagram for the basic object types involved in a grid-based environment is displayed in Figure 5 (this is a conceptual model and not an attempt to describe the actual implementation of the two tools described here).

2.3.6 Visual Programming

Sanscript[7] from Northwoods Software is an example of a visual programming environment based on dataflow. Programs are created by adding objects to a visual function called a flowgram. Objects can be constants, control structures, forms or functions that can be selected from a component library. Each object has a set of inlets and outlets, and objects are visually connected through lines from inlets to outlets (see Figure 6). The function of the object will execute as soon as all data is available on all inlets. A set of objects can be wrapped up to define a reusable component, or just be visually packaged inside a flowgram to make it tidier. Libraries of components called cabinets can be imported into the tool. In the professional version of the software it is also possible to directly import COM type libraries, allowing extension and integration with other applications. Sanscript is aimed to be used by domain experts who are not programmers, but the suitability for this purpose is probably depending on the availability of suitable high-level domain specific cabinets.

Prograph[8] from Pictorius is another example of an environment based on dataflow. The system is object oriented, and requires that the user has some understanding of basic object-oriented concepts, like classes and methods. Methods are created by visually adding constants, operations and conditions etc, and connecting the root of such items to terminals of others using “datalink” connections. The system also allows use of external libraries, such as Windows DLLs.

Dataflow has also been the basis for more domain specific visual programming environments. A prominent example of this is LabVIEW[9] from National Instruments. LabVIEW is targeted at allowing engineers and scientists to create measurement and control applications, with good support for data acquisition, data analysis and testing.

[7] http://www.nwoods.com/sanscript/ (accessed 22. Sept 2005)
[8] http://www.tritera.com/prograph.html (accessed 22. Sept 2005)
[9] http://www.ni.com/labview/ (accessed 22. Sept 2005)

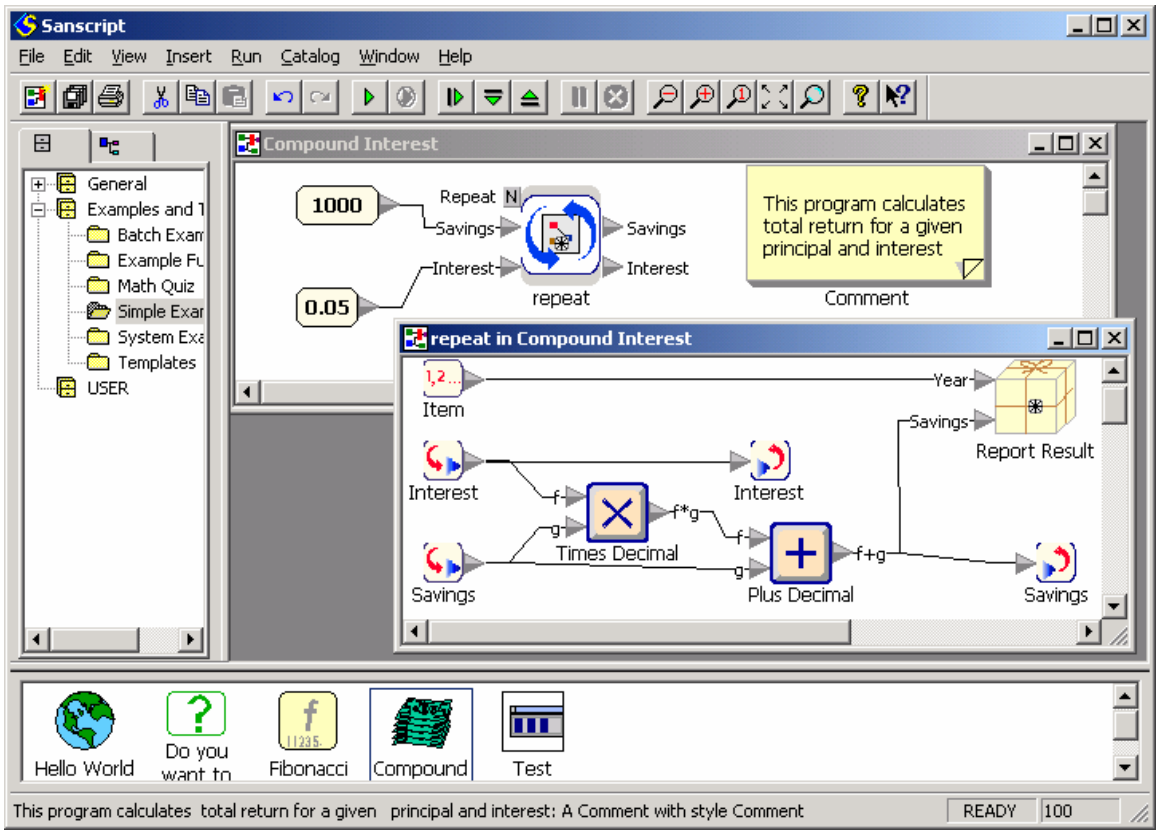

Figure 6 - Compound interest example from the Sanscript environment

Flowcharts have long been used as a way to visualise software programs. Some end-user programming approaches have adopted flowcharts as a way to create programs. The most widespread of these are currently the Lego Mindstorms[10] software that allows children (of all ages) to program robots built from Lego. A program is created by placing program pieces into one or more sequences, each of which will be a separate process. Each sequence can be connected to a triggering condition, for instance, that a touch sensor was pushed, or that a light sensor found a value in a specified range. Within each sequence it is possible to place program pieces representing conditions and loops. Other program pieces are commands that control the speed and direction of connected motors and timing. The system is a closed world, but allows some standard extensions, e.g., to allow new sensors. While the system works fine for small examples, it does not scale up well. Some of the limiting factors of the system are also due to the limitations of the robot hardware – it has quite limited memory, processing ability and also supports only a few sensors and motors.

Updated status on environments

Since the time this section was originally written, Northwood have decided to not make further updates to Sanscript, and are now offering free download and use of the last professional version. Similarly, further commercial development of Prograph has also been stopped. Although this makes these environments less interesting for use in new

[10] http://www.legomindstorms.com/ (accessed 22. Sept 2005)

projects, they remain state-of-the-art examples of visual programming environments, and the description is thus maintained here.

2.3.7 Other Approaches and Research

Spreadsheets

Spreadsheets are so far the greatest success story of practical application of end-user programming. From the success of the first spreadsheet, VisiCalc[11], which was published in 1979, spreadsheets have further evolved and spread to be used by millions of users daily.

As noted by [Johnson 1993] the main limitation - and strength - of spreadsheets is that the range of applications that spreadsheets are well suited for are limited to applications that do calculations over numbers in a table. By using the rows and columns of the table as the single structuring element, the spreadsheet structures are kept simple and understandable to users, and all data and formulas are visible and editable to the user through this structure. Important for the initial success of the spreadsheets was also the fact that the table structure closely resembles paper-based accounting practice.

The formula language of spreadsheets also has a close coupling to the visual organisation of the data. The language is kept simple in structure, with task specific operations that operates of ranges of cells. In this way, more complex programming structures such as loops are avoided. The automatic execution model of the spreadsheet also simplifies the programming, as the user does not need to consider the ordering of calculations.

Compared to the other end-user programming environments, the spreadsheets have some visual similarities with the grid-based environments. These similarities do not go very far though. The differences in execution model, languages and their functions, and types of cell content allowed also results in the differences in the tasks the environments are suitable for.

ACE

[Nardi 1993] proposes task specific visual formalisms as suitable environments for end-user programming. The background for this are user studies which conclude that users without programming background are able to deal with advanced formalisms as long as these are task specific. Examples of such formal systems that people seem willing to learn and handle without much difficulty when interested in the topic is knitting patterns and baseball scoring charts. Generic computer applications tend to support only the least common denominator of the objects and operations that a user may need for a task. Task specific tools can be simpler to use, and better support the task than a more generic tool. As an example, a generic text editor can be use to create slides, but a slide editor is a better tool for the task. [Nardi 1993] further notes that creating task specific environments is time-consuming, and suggests the use of tools for this purpose.

[11] http://www.bricklin.com/visicalc.htm (accessed 22.Sept 2005)

ACE [Nardi 1993] [Zarmer 1992] [Johnson 1993] is an example of such a tool, building on an approach which integrates the application builder with the user's editor by providing a hierarchy of specialised editors that can be further specialised using, e.g., task specific extension languages. In this way the distinction between builder and editor will be blurred, and the final application is the current contents of the editor/builder. A reversal of the traditional development roles are further suggested, where the end user builds as much as he/she can and only draw on the expertise of programmers to add small, more advanced pieces. This is similar to what has frequently been seen in the spreadsheet world.

ACE consists of three layers:

- *ACEKit* serves as the architecture for ACE, and defines policies, protocols and a class library that makes objects builder-readable. The ACEKit adds a basic reflection mechanism which allows any C++ class to be usable from the builder without modification of the builder. Objects have runtime-readable descriptions of operations that can be invoked and of accessors that can retrieve values. These features are similar to the reflection support later provided in component models such as JavaBeans.

- *Visual formalisms* are frameworks for creating visual editing environments, and thus is the central structuring mechanism in ACE applications. Visual formalisms are components that keep appearing across many different applications. They are inherently visual with a clear set of rules governing their editing. Examples are tables, graphs, panels, maps. Four key features architectural features of visual formalisms are that they are interactive, editable, support task-specific content objects, and support relationships between content objects. Visual formalisms can be the main structuring builders, but can also be components used in other visual formalisms.

- *Extension languages* allow the user to add behaviour to the content objects. The formula language of spreadsheets is the best known example of extension languages. The ACE extension language uses the reflection information from ACEKit at runtime to find information about the objects and their operations. When new visual formalisms are linked into the system, they are in this way immediately available to the extension language.

Combined, the features of these three layers make the ACE system overcome the limitations of environments based around a single visual formalism, such as spreadsheets. This makes ACE applicable to a broader class of problems than spreadsheets.

Natural Programming

The goal of the Natural Programming Project at Carnegie Mellon University [Myers 1998] is to develop approaches to programming that makes it easier and more natural to create programs for people who are not professional programmers. Historically, usability has not been emphasised when new programming languages have been developed. Programming in state-of-the-art programming languages may thus be more difficult than necessary, especially for beginners. Programming can be seen as the

transformation of a mental plan in familiar terms into one that is computable. The closeness of the mapping from plan to computable form affects the difficulty of this transformation. The approach taken in the Natural Programming Project is to draw on the results from prior empirical studies of programming along with new studies of how non-programmers express their solutions to programming problems, where vocabulary and structure have been compared with modern programming languages.

HANDS [Pane 2002] is a programming environment for children developed within the Natural Programming Project, where usability is the prime design focus. Based on initial user studies, HANDS selected an event-based structure. Iterations are one of the main obstacles to novice programmers, and HANDS uses aggregate operations on sets of objects as an alternative that simplifies many operations where traditionally a loop would be required. Instead of using explicit data structures, queries are applied. An example including a query and an aggregate operation: *set the nectar of all flowers to 0.* In a traditional programming language this operation would require a data structure to hold "all flowers" and a loop construct iterating over the data structure operating on current object of each iteration. A user study with the HANDS system shows significantly better results with a full version of the system compared to a reduced version where the user had to use more traditional programming constructs.

2.4 End User Tailoring

End user tailoring of software is an area that is related to end-user programming, and where end-user programming techniques can be applied. This sub-chapter briefly presents relevant research in end user tailoring and tailorable software.

[Mørch 1997] (page 3) defines end user tailoring as "*… the activity of adapting generic computer applications to local work practices and user needs.*" In his thesis three levels of tailoring are defined:

- *Customization*. This usually takes the form of preferences forms, and affects the user interface part of the application.

- *Integration*. This level allows users to add functionality to the application, but without full access to the underlying implementation code. End user approaches like macros and scripts is on this level. Also the *rationale* (see below) of the application is placed on this level.

- *Extension*. At this level functionality can be added on the level of the original implementation. The original source code is available, and functionality is added through sub-classing.

Further the concept of application units are introduced, and defined as the "*…smallest self-contained units to be useful in the design and implementation of end user tailorable applications*" [Mørch 1997] (page 53). An application unit contains three parts (or, rather three *aspects* of the application unit, as the term *part* indicate a part-whole hierarchy [Mørch 2002]):

- *Presentation object*. The presentation object is what is directly visible to the user through the user interface, and is the users handle for tailoring. It is modified through *customization*.

- *Rationale*. The rational is related to both the presentation object and the implementation code. It can include instructions on how to use application, what it should do, and why it does it. The rationale is not interpreted by the computer, and can consist of text, pictures, etc. The rationale bridges the gap between the user interface and the implementation code. Rationale is modified through *integration*.

- *Implementation code*. This is where the implementation of the functionality of the application is done, in a suitable programming language. In the tailoring process it can be modified through *extension*.

The concepts of application units and tailoring levels where realised in the tailorable drawing program BasicDraw. In this realisation each application unit had additional event handlers for customization, rationale and implementation. The programming language Beta was used for the implementation, and extension was done through Beta sub-patterns. The program was evaluated with a set of users with some programming experience, who had at least taken an introductory course in object orientation. The study concluded that these users were able to make extensions to the application, and that the intermediate rationale level was helpful during this work.

[Stiemerling 1998] discusses how to create a tailorable component architecture for computer supported cooperative work (CSCW) systems. Here, a tailorable software system is defined as *"… a system, which can be appropriately adapted to changing or diversified requirements."*. This approach is based on computational reflection [Maes 1987], which provides a good basis to achieve the adaptability required to create a tailorable system (reflection is further described in Section 3.2.1). The tailorability of the system is further determined by the choices made for the following issues:

- *Self-representation*. Only aspects of the application that are included in the reflective information can be tailored.

- *Causal connection*. When do changes made to the self-representation take effect? E.g. immediately, after a restart of the system, or after a recompilation of the system.

- *Manipulation functions*. How is the current state of the self-representation presented to the tailor, and which functions are available to change the state?

For systems based on component architectures, three kinds of tailoring are identified. These are:

- *Changing parameters of a single component*. Components can offer a set of different behaviours which can be selected among by changing parameters of the component, for instance, choice of background colour.

- *Changing the composition of components*. These changes can range from simple replacements of a single component, to full rewiring of the component structure (which requires a composition language).

- *Changing or extending the implementation of components*. Depending on the realisation technology, components may be extended by sub-classing the components class. These changes usually require more in-dept programming expertise, and are unlikely to be performed by regular end users.

[Demeyer 1997] discusses design guidelines for tailorable frameworks. Guidelines are proposed to deal with the inherent conflict between reuse and tailorability. The guideline help to identify the hot spots where tailorability is necessary and to specify framework contracts used to formalise which parts of the framework are to be reused. The guidelines are based on identifying the axes of variability for the system being designed, and based on these axes introduce changes to the original design which support interoperability, distribution and extensibility.

2.5 Summary

This chapter has presented an overview of end-user programming research and environments. End-user programming can include creating completely new applications, modifying existing applications and integrating different applications, and is defined as:

- The activity of creating, adding to or modifying the functionality of software, in a way that can be performed without professional programming education and training

The developer and user involved in creating and using end-user programming environments was divided into the roles of *end user* of the finished software, *application developer* who creates the application using the environment, and the *environment developer*.

The main section of the chapter presented a set of end-user programming approaches, with examples of environments and research from the approaches. The presented approaches included general purpose programming languages, scripting, macro recording, programming by example, grid-based environments, and visual programming. A summary of some other approaches and research was also given, before a brief section on end user tailoring and tailorable software research was presented.

Chapter 3 Software Components and Reuse

This chapter presents the state-of-the-art in software reuse and component-based technologies. Reuse is a key issue in achieving quicker development of software, and thus also to addressing the research problem of this thesis: "*How can we streamline and reduce the development effort required to build extensible application composition environments for end users?*". Software component technology is the main foundation for the approaches to reuse and extensibility used in this work.

This chapter starts with an introduction of software reuse in general. Object orientation, which introduced some important reuse techniques, are presented next, and is followed by a presentation of software components and how they relate to object orientation and reuse. A section on software architecture and patterns follows this before component frameworks are introduced. The mainstream object and component models are briefly presented, with main focus on Java and Java Beans as this technology is used as an example of a target platform for the frameworks and tools presented later in this work. A brief presentation of visual component environments is also included.

The chapter closes with a section on user and developer roles involved in component-based software development. This section is presented last because the roles are easier to describe with the rest of this chapter as background.

3.1 Software Reuse

The NATO Software Engineering Conference in 1968 [Naur 1969] can be regarded as the origins of software engineering as a scientific field [Krueger 1992], and is famous for its official recognition of the "software crisis". Already from the start, software reuse was identified as a way to overcome this crisis. At the same conference, the idea of mass-produced software components [McIlroy 1969] as a way to industrialise the software industry was presented. These components, it was proposed, could be organised in parameterised families, and allow the developer to pick and choose components based on, e.g., precision, robustness and generality requirements. Compared to current component terminology, the components envisioned in [McIlroy 1969] were very fine grained – the main example is a family of sine-routines.

[Krueger 1992] gives the following definition of software reuse:

"*Software reuse is the process of creating software systems from existing software assets rather than building software systems from scratch.*"

The main motivation for software reuse is that it will lead to more cost efficient software development and improved quality. Less effort is required to build new software if substantial parts of it can be reused from existing software. The reduced effort required also can lead to reduced time to market for software products.

Software reuse can also be a means to improve the quality of software. When software is reused repeatedly, it gives additional room for improvement and testing of reusable parts as the cost for the improvements and testing can be amortised over a set of reuses. Standardisation and interoperability is also enhanced through software reuse.

Reuse has not revolutionised the software industry as many had hoped for [Sindre 1995]. Partly expectations have been too high, but there have also been technical and organisational barriers. Research has often focused on technical approaches, and often these approaches has been too advanced, requiring more dramatic change to development process than the organisation is prepared to make. [Sindre 1995] and [Karlsson 1995] proposed a more holistic approach to reuse, with support for both technical and organisation aspects. Reuse has two fundamental aspects that needs support both on the technical and organisational level:

- Development *for* reuse: development of reusable components.
- Development *with* reuse: development of systems from existing components.

[Reenskaug 1996] argues that both creation and application of reusable components depend on appropriate solutions along the three software engineering dimensions of technology, organisation and process. An industrial approach to software production should be organised as a value chain, where people at one level build on the results from the layer below, and deliver results to the layer above.

Both [Reenskaug 1996] and [Sindre 1995] identify the organisational issues as the most challenging issues to achieve reuse. Management has to be convinced that reuse is worthwhile, and to understand and accept that it requires some additional initial investment and suitable and stable organisation. Also, developers have to be convinced, battling the not-invented-here syndrome. Although reusability can to some degree be designed from the start, a fundamental insight is that "*before you can reuse something, you must use it*" (according to [Reenskaug 1996], page 134, Brad Cox is the source for this statement). For the reuse development process, this means that reusable assets building (at least partly) is a reverse engineering activity, where products built using a forward engineering approach is analysed and reworked for future reusability.

Although components are often regarded as the main reuse technology, it can be argued that reuse also can take different forms that may be even more successful. [Krueger 1992] claims that reuse technology also includes, for instance, application generators and very high level languages. High level languages can even be seen as the greatest success story of reuse so far. Reuse technology can be classified based on their support along four dimensions: form of *abstraction* for software artefacts, *selection* of reusable

artefacts (e.g. locating and comparing), *specialisation* from the generic artefact to the use situation (e.g. parameters, transformations), and *integration* to create complete software systems. Raising the abstraction level for software engineering has proven to be difficult, and this is part of the reason why reuse also is difficult.

Some of the main insights of [Krueger 1992] are summarised in the following "*truisms*":

- "*For a software reuse technique to be efficient, it must reduce the cognitive distance between the initial concept of a system and its final executable implementation*".
- "*For a software reuse technique to be efficient, it must be easier to reuse the artifacts than it is to develop the software from scratch*".
- "*To select an artifact for reuse, you must know what it does*".
- "*To reuse a software artifact efficiently, you must be able to find it faster than you could build it*".

We return to the issue of reuse in relation to software components in Section 3.3.2.

3.2 Object Orientation

The basic idea of object-orientation is to model our programs based on entities that mirror the real world, where each entity contains its own state and behaviour.

The roots of object orientation go back to the Simula-67 language developed at Norsk Regnesentral for simulation of real-world phenomena. The ideas were picked up by researches at Xerox Parc, and further developed with their Smalltalk language and environment. The more widespread acceptance of object-orientation came with the C++ language in the late 1980s, and with Java in the late 1990s.

The most widespread general programming languages based on objects all use objects, classes, and inheritance, and thus fits the definition in [Wegner 1987] of an object-oriented language. Other approaches also have been attempted, e.g., with prototypes instead of classes like the Self language [Ungar 1991], but is not in widespread use.

Scripting languages based on objects are also now in common use. These languages, like JavaScript and Visual Basic for Applications, have a broader user group than the general programming languages. The use of objects in these languages is focused on use of exiting object types, and thus the languages usually provide little support for the definition of new classes and especially in the use of inheritance. In the terminology of [Wegner 1987] these languages usually qualify as object-based, but not object oriented.

Object orientated languages usually support different kinds of software reuse. [Gamma 1994] (page 19) defines two kinds of reuse; black-box reuse and white-box reuse. In the first of these, existing classes are regarded as black-boxes, and encapsulation are used to ensure that no internal parts of the classes are (considered) known. Reuse is done by object composition, where the instances of the class are created and used as-is, based only on the public interface provided by the class. White-box reuse is primarily done by

subclassing. Parts of the internal structure of the parent classes are often visible to subclasses, allowing the subclass to utilise implementation details that are encapsulated from normal "clients" of the parent class. In most languages a subclass can override the definition of a method from its parent class by its own version, and can call the parent's method when required. This allows extensions and modifications to the original behaviour of the parent class. Other approaches also exist: the Beta language, e.g., defines an inner construct which allows a parent class to explicitly define points in a method where overriding behaviour of a subclass method will be called, enabling the parent class method to wrap the execution of subclass method.

Modern object-oriented languages usually support the separation of interface from implementation. This enables the construction of systems that can be extended with classes that support a required set of interfaces. In earlier approaches interface was always mixed with a class definition, and extensibility was limited by requiring the use of implementation inheritance from the parent class defined by the extensible system.

3.2.1 Reflection

Computational reflection is a technique that allows a system to perform computations about itself, and thereby possibly modify itself. This ability of a system can be used to support more dynamic extensibility and adaptivity. [Maes 1987] give the following definition:

"*A reflective system is a system which incorporates structures representing (aspects of) itself.*"

A reflective system has a casually connected self-representation of itself, and thus [Maes 1987] observes that:

- "*The system has an accurate representation of itself*".
- "*The status and computation of the system are always in compliance with this representation. This means that a reflective system can actually bring modifications to itself by virtue of its own computation*".

The value of computational reflection has been widely recognised. Modern programming languages as well as distributed object and component models are increasingly built on a reflective architecture. The degrees of reflection support varies though, from only providing non-modifiable information such as which methods a class provides, to allowing modifications that affects the computation. Even limited forms of reflection can allow building of high level tools (such as class browsers), and increases the extensibility as important aspects of an object can be discovered and utilised at runtime.

3.2.2 Extensions to Object Orientation

While object-orientation is widely regarded as state-of-the-art in software development, there are still problems that do not have clean solutions with the current object-oriented technology. In the last years there has been a growing discussion about such problems, and various solutions has been proposed.

Aspects

One of the new approaches that have emerged is aspect-oriented programming. [Kiczales 1997] claims that object-orientation is good at expressing the base functionality of a system, but some other design decisions are difficult to cleanly capture in code. These *aspects* (e.g., synchronisation, error handling, constraints and optimisation) crosscut the base functionality of the system, resulting in tangled code. The proposed solution to this is to code the aspects separate from the base functionality, and use a "weaver" to merge the code into a solution.

AspectJ [Kiczales 2001] is an aspect-oriented extension to Java that allows developers to express aspects in a modular way. The AspectJ language supports two different kinds of crosscutting implementation. Static crosscutting makes it possible to define new operations on existing types, and thus affects the static type signature of the program. Dynamic crosscutting allows the developer to add behaviour that will be executed at specified well-defined points of the execution of the program. Such execution points, called join-points, include method call reception, and get- and set-operations on fields of an object. Pointcut designator primitives of the language allow the developer to identify a set of join-points, and these primitives can be combined to express more specific sets, e.g., all calls of a specific method which originates outside a specific class. A method like mechanism, called advice, is used to specify code that should be executed at the join-points in a pointcut. Aspects are class like constructs, which allow the developer to express a modular unit of crosscutting implementation. An aspect can contain pointcut and advice declarations, as well as other declarations normally found in a class.

Hyperspaces

One of the goals of object-orientation is to achieve simplicity through separation of concerns. The concerns are decomposed into classes, but some concerns like features and aspects do not map well to classes, and causes scattering and tangling of code. [Ossher 2001] criticises this "tyranny of the dominant decomposition" that object-orientation is one example of, and proposes the Hyperspace approach to address multi-dimensional separation of concerns. Requirements to a separation of concerns mechanism are:

- it must be possible to encapsulate any dimension of concerns simultaneously, and each dimension must be created equal;
- the developer must be able to identify new concerns incrementally, at any point of the software lifecycle;
- the developer can handle each concern without knowing about concerns and dimensions that do not affect the current activity directly;
- it must be possible to represent overlapping and interacting concerns;
- there must be a powerful integration mechanism to integrate the separate concerns.

Hyper/J is a tool that supports hyperspaces using Java to define the units, and is an attempt to address these requirements. According to [Ossher 2001] it allows incremental adoption; it provides useful features also when used to extend existing Java software.

3.3 Software Components

From the middle of the 1990s components became a frequently used term in the software community, but there was no real common foundation or definition of the term. A major theoretical contribution to the field of software components was made by Clemens Szyperski in his book "Component Software" [Szyperski 1997], and this work has since been regarded as a foundation of the field in the academic community. This chapter is greatly influenced by Szyperski's book.

The definition of a software component used in this thesis was made at the Workshop on Component Oriented Programming at ECOOP 1996, and is also found in [Szyperski 1997]:

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

Below follows a more detailed discussion of the elements of this definition.

- *Contractually specified interfaces*. These interfaces specify the services the component can perform. In current component technologies the interfaces are usually defined in and Interface Description Language (IDL). Current IDL versions are usually limited only to the syntactic definition of interfaces, while semantic issues such as preconditions and postconditions, and quality-of-service (QoS) specifications are not included. From a formal point of view semantics should clearly be part of the contract, but from a more practical view the need for formal definitions should be weighted towards the additional cost and effort it requires.

- *Explicit context dependencies*. A component can depend on the interfaces of other components. When should these requirements be checked? This situation is somewhat similar to other early/late binding problems, with some of the same pros and cons. A check may earliest be performed at component deployment time. One reason for performing the check this early is that the deployer can get immediate feedback. Delaying the test until runtime gains some flexibility, but a failure situation may then become visible to the end user of the system instead of the component deployer. Current IDL's are focused on describing the interface of the component, while ignoring the context dependencies. This is a clear weakness of current approaches.

- *Unit of composition*. Components are units of composition that makes sense to threat as a whole. A component is usually made because it is believed to be a unit that will be reusable in different situations. Another reason for making a component out of something may be because it gives a cleaner solution to the problem at hand, and is thus worthwhile even without further reuse. As the unit of composition it is also natural to expect some kind of support in the component technology of choice for making composites with it, e.g., in component composition languages.

- *Independent deployment*. A component should be independently deployable. This does not necessarily mean that the component is always deployed separately – it will often make sense to deploy a solution created from many components as one unit. Partial deployment of a component is normally not possible or desirable. It may perhaps be argued that if hierarchical compositions of components are regarded as components, it would technically be possible to deploy parts of the composite. A closer look at this reveals that a motivation for doing this would either be to have a choice of alternate compositions, which are really alternate components, or to install sub-compositions that should be regarded as components in their own right.

- *Composition by third parties*. Reusable components are often intended for use by third parties. Although there has been an increasing tendency to provide source code of products also to third parties, it is expected that most components primarily are available in binary format. This means that the components will often be black-box, and thus increases the importance of the specification of interfaces and context dependencies. Currently the users of such components have to rely on non-formal documentation that is provided in addition to the specifications, as semantics are not covered by the specification.

Software components according to this definition are good basis for building extensible systems. Contractually specified interfaces and independent deployment open for components that can replace or add to initially provided functionality of a component-based application.

A unique property of software that makes software components different from components in other disciplines is described in [Szyperski 1997]. "*Rather than delivering a final product, delivery of software means delivering the blueprint for products*". This cause some problems when we try to compare components used in other disciplines to software components, and both "plans" and "products" are frequently referred to as components. Usually a software component allows multiple instances of objects to be created from it.

3.3.1 Software Components and Objects

Much of the work on components has been done in an object-oriented context. Although components and object-orientation can be seen as a natural match, components can also be made and used outside of object-oriented contexts. Applications running on top of modern operating systems can, e.g., be regarded as very coarse grained components [Szyperski 1997], independently on how they are implemented.

In an object-oriented setting, a component can contain a set of classes and static objects, along with other resources. Each class is fully contained in a single component, but its superclass may be part of another component. The interface of a component is defined by a subset of the classes and static objects it contains. Although a component can contain multiple classes, there can, for instance, be only a single class that is visible from outside the component with the rest of the classes being used only as part of the private implementation of this class.

3.3.2 Software Components and Reuse

Components are used as units of reuse in many engineering disciplines. The success of integrated circuits (ICs) has been a source of inspiration for the early development of software components, but the analogy to ICs gives limited insight into reuse of software components due the blueprint properties of software [Szyperski 1997].

Software components can be reused using both black-box and white-box reuse, as described in Section 3.2. When used as black-boxes, the components are used "as-is". In an object-oriented setting, this usually means that the classes provided by the component are used to create object instances, either by other components or by applications. In white-box reuse, the classes provided by one component are reused through subclassing by other components or applications. [Szyperski 1997] discusses some problems with inheritance and components: if a new release is made of a component containing a superclass, it may break subclasses in other components. Proper use of private and protected mechanisms in modern languages can partly avoid the problem. Researchers has suggested the use of specialisation interfaces and reuse contracts to further formalise the contract between a superclass and its subclasses, but so far the main programming languages have not adopted these approaches.

[Bosch 1999] discusses techniques for adapting components, addressing the problem that "as-is" reuse of software components is unlikely to occur. There is a set of requirements for component adaptation techniques. The adaptation should be *transparent* for both clients and the component itself, and be based on *black-box* reuse. Further, adaptations should be easily *composable* with the component requiring no redefinition. Adaptations should also be *configurable* and *reusable* by separating their generic parts from their configurable specific parts. Traditional adaptation techniques like copy-paste, inheritance, aggregation and wrapping fulfils few of these requirements. In the *superimposition* adaptation technique proposed by [Bosch 1999] components and the functionality adapting components are two separate entities, gaining reusable components and reusable adaptations. Adaptation types supported by the Superimposition include changes to component interface, component composition, and component monitoring. Superimposition has some similarities with aspect-orientation (see Section 3.2.2) both in insisting on two separate entities for components and adaptations, and in some of the problems it attempts to address.

Selection of black-box versus more open nature components can also have impact from an economical viewpoint. Commercial-Off-The-Shelf (COTS) components are typically provided in a black-box, binary form without access to source code. COTS have obvious benefits compared to internally developing the functionality, such as "*…the functionality desired can be: (1) accessed immediately, (2) obtained at a significantly lower price, and (3) developed by someone who is an expert in that functionality*" [Voas 1998]. However, as [Voas 1998] further points out, these benefits have to be weighted against the economical risks involved, e.g., due to reimplementation or workarounds required in the event that the COTS component provider should go out of business or discontinue the development.

In a recent empirical study of Off-The-Shelf (OTS) components usage in industrial projects [Li 2005], shorter time-to-market, lesser development effort, and to get the newest technology were the main motivations for using OTS components. [Li 2005] further studied why decision makers choose COTS components instead of Open Source

Software (OSS) components, or vice versa. While greater trust in quality and support ability were the main points in favour of COTS vendors, possible free source code and access to revise the source code if required were points in favour of OSS. Results from using OTS components further showed that relative to each other COTS users had more problems in following requirement changes and in controlling negative effects on systems security, while OSS users were more uncertain on the OSS provider support reputation.

3.4 Software Architecture, Patterns and Frameworks

The research area of software architecture has emerged over the past decade, and from its roots in qualitative descriptions of system organisation the area has evolved to include notations, tools and analyses techniques [Shaw 2001]. There are numerous definitions of software architecture found in the research literature, but here the [IEEE 2000] terminology is adopted, which defines architecture as:

"The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution".

As [Szyperski 1997] points out, an overall architecture provides the firm foundation required for the evolution and maintenance of components and component-based systems. The architecture should categorise central resources to ensure the independence of components competing for the same resources. The architecture defines the roles of the central components and their collaboration, along with overall invariants of the system, and establishes the points of extensibility of the system.

In addition to the functionality of the system, the architecture also determines quality related attributes of the system, such as performance, reliability and security. [Bosch 2000] defines a quality attribute-oriented architectural design method. In this method, the initial application architecture is derived from a functionality-based architectural design. The quality attributes of the architectural design are then estimated, and if the design does not satisfy the quality requirements, a set of transformations for optimising specific quality attributes are selected and applied to derive an updated design. Transformations tend to optimise some quality attributes while affecting other attributes negatively, and the estimation and transformation steps are repeated until a suitable design is achieved or until the architect have to give up (and probably renegotiate the quality requirements).

Software patterns are a reuse mechanism applicable to both architectural and detailed design of software, and are based on the idea of reusing design solutions that has proven to work well. The idea of using patterns in software was adopted by [Gamma 1994] from [Alexander 1977] which originally described the use of patters in buildings and towns. The basic idea of patterns is well expressed in the following quote from [Alexander 1977] (page x), and is as relevant for software as for its original domain:

"*Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice*".

In the context of object-oriented software, [Gamma 1994] has the following definition of design patterns:

"*A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementations hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context.*"

[Gamma 1994] defines a template for describing software design patterns, and provides a catalogue of patterns. The patterns are categorised in the two dimensions of *purpose* and *scope*, where purpose can be creational, structural or behavioural, while scope is either class or object. The patterns in the catalogue of [Gamma 1994] all refer to one or more implementation they occur in, following the principle that patterns are not "created" but rather discovered from well tried solutions.

Other catalogues of design patterns have later been published following similar principles as [Gamma 1994]. [Buschmann 1996] and [Schmidt 2000] are two good sources of patterns applicable to architecture. [Buschmann 1996] uses the following definition for architectural patterns:

"*An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.*"

An object-oriented framework can be defined as ([Reenskaug 1996] page 17):

"*... a collection of classes that together solve a general problem, and that is intended for specialisation through subclassing.*"

When comparing patterns and frameworks, patterns are more abstract, are smaller architectural elements and are less specialised than frameworks ([Gamma 1994] page 28). While a pattern tells how to solve a problem, a framework contains a "canned solution" ([Reenskaug 1996] page 17). An object-oriented framework often contains realisations of many patterns.

Application frameworks usually predefine the main structure and generic behaviour of an application, and allow the application developer to fill in the application specific parts. In these frameworks, subclassing has traditionally been used to minimise the work of the application developer. The developer typically implements methods which the frameworks mark as abstract, and overrides other selected methods to implement the application specific behaviour. The main flow of control in the application is usually handled by the framework, sometimes referred to as a "don't call us – we call you" philosophy.

3.5 Component Framework

Software components as defined in Section 3.3 and supported by the wiring standards provided by standard component models, is a necessary foundation for building

extensible systems. However, to ensure that a set of independently developed components extending a system will collaborate usefully with each other and the system, a further set of rules is required.

[Weck 1997] defines an independently extensible system as a system that allows functionality to be added at runtime, and where extensions can be developed independently by third parties. Independent extensibility requires strict rules, as the developers of different components do not even know about each other. A component framework is a set of rules to be obeyed by components in particular environment.

An area where extensibility and independently developed components have been a success, although mostly at design-time, is user interfaces (e.g., Visual Basic). One of the main reasons for this is that there is a set of rules, in some cases enforced by a framework, for how user interface components should behave. As the set of rules is related to a visual formalism, it is also easy to envision for the developer. A user interface component can expect from its environment that it is (usually) included in a container, will receive events, is told when and where to paint itself, and is governed by a layout policy. The component is typically expected to paint itself, keep track of its position, take care of container responsibilities if it is a container, and react to events.

User interface environments supporting such extensibility by independently developed components can be regarded as (simple) component frameworks. [Szyperski 1997] has the following definition:

"*A component framework is a dedicated and focused architecture, usually around a few key mechanisms, and a fixed set of policies for mechanisms at the component level.*"

The key contribution of a component framework is that it partially enforces architectural principles [Szyperski 1997]. This greatly increases the chances that independently developed components can usefully collaborate with each other and the system. For the component developer a component framework simplifies the development, as it makes it clearer what the component can expect from its environment and what is expected from the component. A component developed without such a framework as reference will usually require more adaptation. On the other hand, a component developed for a specific component framework is not likely to be reusable outside this framework.

While an application framework partially implements a functional design, the main task of component frameworks is to enforce design rules [Weck 1997]. A component framework does not necessarily implement any functionality, but may, for example, implement resource management and support communication between components. A component framework defines a set of dimensions of extensions. As an example, a compound document component framework can support the dimensions of containers and document parts. Components along the same dimension can be called parallel extensions as they play the same role and thus can be treated equally by the framework. Orthogonal extensions serve different purposes in the framework, and are used to separate concerns. Orthogonally extensible component framework defines interfaces for interaction between components from different dimensions (e.g., container and document part).

Building an application as a component framework allows the application to define its own runtime variability and extensibility. When combined with an explicit runtime representation of the application's architecture model and annotation of components with properties characteristics, this can enable a system to adapt its architecture at runtime to maintain architectural qualities in a changing, mobile environment [Hallsteinsen 2005] [Floch 2006].

3.6 Object and Component Models

This section presents the mainstream object and component models. CORBA and COM are first briefly described. As Java and Java Beans is used as an example target platform for the component frameworks and tools proposed later in this work, these technologies are described a bit more in detail. The motivation for selecting Java and Java Beans is discussed in a Section 7.2.

3.6.1 CORBA

CORBA [OMG 2004], or Common Object Request Broker Architecture, is a vendor-independent architecture for distributed applications. CORBA is one of the standards managed by the Object Management Group (OMG). OMG is an open membership, non-profit consortium where most of the leading companies of the computer industry are members. One of OMG's main tasks is the development of the Object Management Architecture (OMA) of which CORBA is a central part.

OMA is composed of two central parts; an object model, and a reference model [Vinoski 1997]. The object model provides a common semantics for specifying the externally visible part of objects in an implementation independent way, while the reference model describes categories of objects and how these collaborate.

The following is a summary of the OMA object model [Pope 1998]:

- *Identity*. Each object has a unique identity (OID). There may be multiple objects of the same object type, but the identities always distinguish these objects from each other.

- *Operations*. Operations are defined by their signature, which consists of name, a set of parameters, a result and a set of exceptions. The consequences of an operation are defined as an immediate result set, side effects (e.g., state changes in the object) and exceptions. Operations are only defined on object types, and not on types that are not objects.

- *Object types*. Object instances are of a specific type. This type can not be changed after the object has been created. An object type is a composition of a set of operations.

- *Non object types*. A non-object is something that does not have an OID. They do not have a set of operations, and are not receivers of messages. Non-objects can be used as parameters of operations.

- *Inheritance*. A type can be a subtype of one or more types. This provides multiple interface inheritance. Implementation inheritance is allowed, but not required in the model.

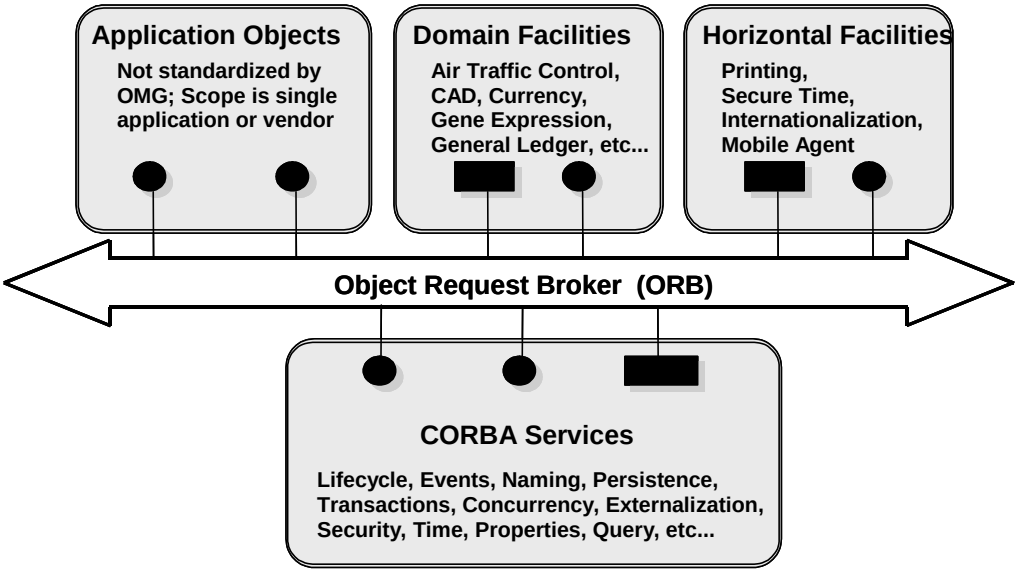

Figure 7 - OMA reference model with examples from the categories of interfaces

Figure 7 shows the most important parts of the OMA reference model. A brief description of the parts:

- *ORB*. This is the infrastructure that lets the objects communicate with each other, and is the "heart" of the communication.

- *CORBA Services*. These are domain independent system level services that provide basic functionality that is frequently used by distributed applications. They can be regarded as a kind of extension of the basic functionality of the ORB. The services give the objects a generic environment that they require to do their task.

- *Horizontal Facilities*. These are also domain independent services, and sit between the CORBA Services and the Application Objects. The services facilitate end user applications and are generally useful for most applications.

- *Domain Facilities*. Vertically oriented services for specific domains; e.g., finance, health, electronic commerce etc. These services standardises central interfaces of the application domains.

- *Application Objects*. Objects specific for the applications that are developed.

CORBA is the specification of the ORB part of OMA. CORBA is the "bus" that allows the distributed objects to communicate with each other.

CORBA objects can be implemented on diverse platforms and in various languages. To allow objects in such a heterogeneous environment to communicate with each other, a common definition of interfaces is required. For this task OMG's Interface Definition

Language (IDL) is used. Standards are defined for how various programming languages are mapped to/from IDL. In this way objects can communicate with each other without knowing the platform and implementation language of the other objects.

Distribution is supported through the use of client stubs and/or dynamic invocation interfaces on the client side, and object adapters usually combined with implementation skeletons on the server side. Client stubs and implementation skeletons are generated from IDL by programming language specific support tools, and hide the interaction with the lower level ORB mechanisms.

CORBA Components

CORBA Components [OMG 2002] is a specification of a component extension to CORBA, and is included in the specification of CORBA 3. The component meta-type is an extension of and specialisation of the object meta-type, and a component specification is an extension and specialisation of an interface definition.

Components can support a variety of surface features, called ports. The different kinds of port types are:

- *Facets*. A component can provide multiple object references with distinct IDL interfaces, called facets.
- *Receptacles*. Receptacles are named connections points. These accept connections to other objects upon which the component may invoke operations.
- *Event sources*. Defines that the component can generate events of a specific type, and allows consumers to be associated.
- *Event sinks*. Defines that the component can receive events of a specific type.
- *Attributes*. Named values exposed by the component, primarily intended for configuration purposes.

The standard defines two levels of components; basic and extended. Basic components are only allowed to offer attributes; extended components can offer any port type. Basic components are intentionally kept very similar to Enterprise JavaBeans, allowing easy mapping and integration.

3.6.2 Microsoft's COM and .NET

COM

COM [Microsoft 1995], or Component Object Model, is Microsoft's architecture for software components. COM is a binary interoperability standard that is independent of programming language.

COM objects are interacted with through the use of interfaces. Each object can implement multiple interfaces. All COM object must have an interface called

IUnknown. This interface contains functions for reference counting, and a function for querying for other interfaces of the object. Interfaces in COM can singly inherit another interface, and IUnknown is the base interface of all other interfaces. In this way the functions of IUnknown is available through any interface pointer. Beside IUnknown there are only a few other interfaces that are frequently inherited from. Polymorphism is supported through the ability to implement multiple interfaces, and not through interface inheritance.

The only access clients have to COM objects are through pointers to interface nodes. An interface node contains a pointer to a table of functions pointers – also called a vtable as it is derived from the tables used to implement virtual functions in C++ [Szyperski 1997].

The client has no single pointer to the whole COM object, only pointers to interfaces. To get hold of another interface of the object, the QueryInterface function of IUnknown must be used. COM breaks with the classical object definitions by not having a unique object ID. Interfaces in COM are immutable. Objects are allowed to evolve over time by supporting new interfaces, maintaining binary compatibility with existing client.

COM is as earlier mentioned a binary interoperability standard. Unlike CORBA there is no *required* common definition language that all implementations must use. A common source level language can still be useful though, and Microsoft's Interface Definition Language based on DCE IDL is such a standard that is *usually* used for describing the COM interfaces [Microsoft 1995].

To avoid naming conflicts there has to be some way to uniquely identify an interface. For this purpose each COM interface has a Globaly Unique Identifiers (GUID). This is a 128-bit integer that is virtually guaranteed to be unique, and is used for all identifications of interfaces. For convenience the interface in addition has a human-readable name; e.g., IUnknown, but these names are not globally unique, and it is always the GUID that is used behind the scene for identification.

Microsoft's Distributed COM (DCOM) is an extension of COM, and is based on the Open Software Foundation's DCE-RPC spec. Interaction with the DCE-RPC is hidden through an approach similar to CORBA. In DCOM terminology proxy objects handle this on the client side, while stubs are used on the server side.

COM is also the foundation for other services such as uniform data transfer, persistent storage of compound files, and monkiers (intelligent names). As Figure 8 shows, these services are further the basis for Microsoft's compound document technology: OLE (Object Linking and Embedding). OLE defines a set of COM interfaces that enable applications to embed or link into their documents content that is not native to the application. The technology supports showing the embedded content, and also allows in-place activation where part of the container's screen estate is handed over to the server of the embedded content.

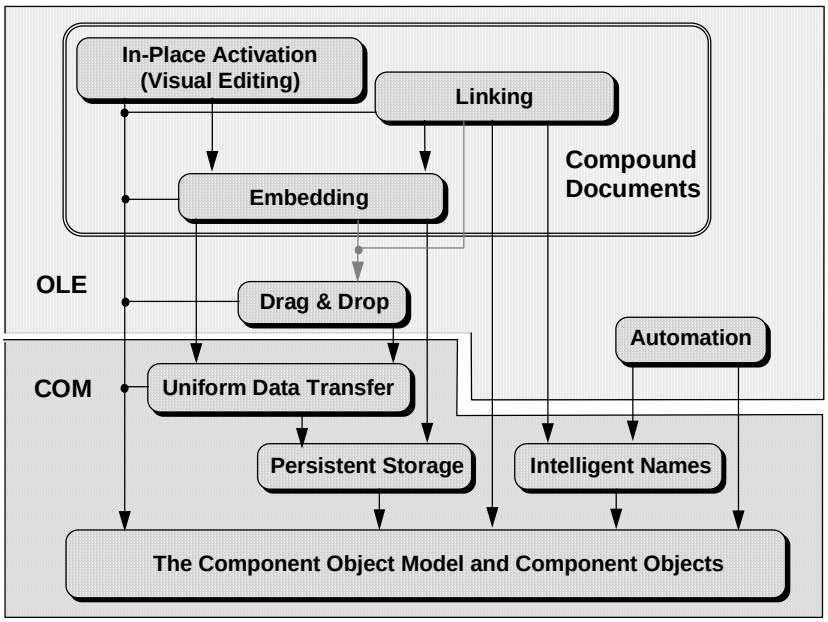

Figure 8 – Relations between OLE and COM services (from [Microsoft 1995])

Automation is a COM-based technology that allows applications to expose their functionality to scripting languages and other applications. The objects exposing their functionality are called ActiveX objects, while the tools accessing the functionality are called ActiveX clients. Applications or libraries capable of creating one or more ActiveX objects are called ActiveX components. ActiveX objects can expose methods that can be called on them, properties that access the state of object, and events that the object can generate. Central in the automation mechanism is the COM interface IDispatch which is used for dynamic invocation of methods and property functions, and also can be used to find ids for property and method names at runtime.

ActiveX controls are client side COM components with design-time UI. It fills the role that OLE controls (OCXs) had before it, and Visual Basic controls (VBXs) had even earlier. The only requirements on ActiveX controls are that they implement the IUnknown interface and support self-registration. One of the problems of OCXs was that they were required to implement a large number of interfaces [Szyperski 1997]. For downloading over the net more lightweight solutions were required. The ActiveX controls technology instead defines many optional features that the control can implement.

.NET

The latest addition to Microsoft's strategic technologies is the .NET framework. Although the framework can be applied to most development tasks, its main focus of attention has been to enable easy development of XML web services with web-based clients. The .NET Framework has two main components: the Common Language Runtime (CLR) and the .NET Framework class library.

CLR provides core services such as memory management (including automatic garbage collection), thread execution, and code safety verification. The CLR has a common type system to which all languages supporting .NET conform. The language compilers emit

metadata that describes the types, members and references used in the code. Through this approach CLR can allow developers to choose any .NET supporting language, and fully utilise both the common class library as well as components written in other languages. A wide range of languages is available, including C++, C#, Java, Component Pascal, Visual Basic, and Smalltalk. Just-in-time (JIT) compilation is used to run the managed code produced for the CLR as native machine language.

The .NET Framework class library contains a wide range of classes, supporting basic features such as i/o, string manipulation, network communication and computational reflection. The framework has strong support for data access and XML handling. Through a set of classes known as ADO.NET, retrieval, manipulation and update of data from various sources such as Microsoft SQL Server, OLE DB, and XML is supported. A central part of ADO.NET is the DataSet object which enables disconnected access to data. XML is used as the native serialisation mechanism for the DataSet.

The framework also contains support for creating different kinds of user interfaces. Windows Forms is the platform for building desktop-based clients based on Windows, while the ASP.NET technology supports building of dynamic web pages using an event-driven programming model.

3.6.3 Java and Java Beans

From its first public release in late 1995 Java has grown to be one of the currently most used programming languages. In addition to being an object-oriented programming language Java is also a software platform (or, strictly speaking, a set of platforms). Java's initial quick spread was largely due to the popularity of Java Applets; small programs that can be embedded in web pages, and that are automatically downloaded and executed on the client computer. Applets operate in a "sandbox" that limits what they are allowed to do, ensuring security for the user.

As a platform, Java in addition has a virtual machine (VM) and an API. With the arrival of Java 2, a set of Java editions was defined:

- *Java 2 Standard Edition (J2SE).* The standard Java edition is a development and runtime platform, and includes a VM and the basic APIs used to make and run Java applets and applications.

- *Java 2 Enterprise Edition (J2EE).* J2EE is based on the J2SE, and in addition defines an environment and includes a set of APIs that is useful for enterprise applications.

- *Java 2 Micro Edition (J2ME).* This edition addresses smaller devices, from smart cards up to handheld computers. A set of profiles is defined, with a VM and API that fits the target device. Applications written for the J2ME are upwardly scaleable to J2SE and J2EE, although code using profile specific APIs, such as for user interfaces, will have to be rewritten.

This subsection describes some of the different Java-based technologies. As Java and JavaBeans is used as an example of a target platform for the frameworks and tools

proposed later in this thesis, the Java object model and the JavaBeans component model is described in somewhat more detail than the technologies presented earlier in this chapter. The motivation for selecting Java and Java Beans is discussed in a Section 7.2. The enterprise edition of Java is also briefly presented. The descriptions here are based on J2SE 1.4 and J2EE 1.4.

Java Object Model

As a language, Java "*is a general-purpose, concurrent, class-based, object-oriented language*" [Gosling 2000]. In the design of the language proven state-of-the-art solutions were selected, and it was attempted to keep the language small and keep a well known style similar to C/C++.

Features of the Java object model:

- *Primitive types*. The Java language has a predefined set of primitive non-object types that includes boolean, integer, floating point and characters. It is not possible to define operations directly on the primitive types, but the language contains a predefined set of operators that can be used (e.g., +, *, etc). The primitive types can be used in variable declarations and as parameters of operations.

- *Objects*. Objects are instances of classes or arrays. Objects have associated operations and variables as defined in their class. Objects are accessed through reference. Reference types are the same type if they have the same binary name, and are loaded by the same class loader.

- *Operations*. The signature of an operation consists of the name and a set of formal parameters. A class may not declare two operations with the same signature. In addition to the signature, an operation has a result type, and can define a set of exceptions and modifiers. Operations are defined on classes and interfaces, but not on primitive types. Operations are defined for object instances unless the operations modifier *static* is used to mark the operation as a class method.

- *Single implementation inheritance*. Java classes inherit the implementation of one other class. All other classes inherit directly or indirectly from the java.lang.Object class.

- *Multiple interface inheritance*. Java allows a class to implement multiple interfaces. This extends the use of polymorphism provided by the implementation inheritance hierarchy.

- *Reflection*. The Java language has some features for introspection and reflection. Class objects are present at runtime, and through these and related classes it is possible to discover the methods, fields and constructors of the class. A generic invoke mechanism can be used to call any method and pass the required parameters. From version J2SE 1.3 a Proxy class is included which supports creating dynamic proxy classes based on a set of interfaces specified at runtime.

Calls to dynamic proxy instances are dispatched to an invocation handler, allowing interception mechanism to be created.

- *Garbage collection*. Java has automatic garbage collection. Objects or groups of objects with no references left to from the rest of the system will automatically be released by the garbage collector. This makes the language easier to use and avoids some cases of memory leakage and also accidental manipulation of memory content through pointers that were released too early. A problem is that it may in some situations result in a performance penalty.

- *Packages*. Classes and interfaces in Java are organised in packages. Classes can be defined as private to a package, and is then not accessible outside the package. Classes and interfaces can import public definitions from other packages, for use in their own definition and implementation. Rules for naming of packages addresses the globally unique naming of classes and interfaces, based on the web domain name of the developer in reversed sequence as the prefix for all the developer's packages.

The Java language is usually compiled to bytecodes; a platform independent code that can be interpreted by a Java VM. This is the basis for Java's for "write once, run anywhere" properties.

Java provides a distributed object model through the Remote Method Invocation (RMI) which is part of J2SE. RMI works in a way that is similar to CORBA and DOM, with proxies at both ends of the communication. Remote method calls are done exactly as any other Java method call, although the caller must be prepared to catch RMI exceptions. The methods must be part of in interface that extends java.rmi.Remote. Remote methods can use primitive datatypes and remote objects as parameters. In addition objects that support the Java serialisation mechanism can be used, and will then be passed "by copy". A special feature of RMI is that when serialisable objects are used and the class is not locally available, the bytecode of the class is transparently loaded from the other end, thus supporting limited code mobility.

Java Beans

JavaBeans is the component model of the Java 2 Standard Edition. In the [Hamilton 1997] specification a JavaBean is defined to be "*... a reusable software component that can be manipulated visually in a builder tool*". This does not imply that the component itself needs to be visual. The only absolute requirements for a JavaBean is that it is a Java class that has a constructor without parameters, and that it implements the empty marker interface Serializable to support persistence. Visual JavaBeans will usually extend the Component class of java.awt, or one of its subclasses.

To be more useful to the developer, the bean should follow some of the standards defined by the JavaBeans model:

- *Introspection*. The JavaBeans model uses the introspection mechanisms of the Java language to make the job easier for the developer. By following simple naming conventions/patterns much of the definition of JavaBean can be derived by the supporting tools.

- *Properties*. The properties of a JavaBean can be defined by following a naming convention; typically the read and write methods are named after the property, and prefixed with “get”/”is” and “set”. JavaBeans properties can in addition be “bound” and/or “constrained”; for “bound” properties the JavaBean sends a message to registered listeners after each change to the property, while for “constrained” properties messages are sent before the change, allowing listeners to veto the change.

- *Events*. A JavaBean can notify interested parties of important events through the JavaBeans event mechanism. The JavaBean can register listeners for a set of event listener interfaces that the interested parties should implement. When an event occurs the registered parties are notified by calls to one of methods in the interface, with an event object providing further information. Patterns are used both for naming the interfaces and defining their methods, and naming the methods that allow listeners to be added and removed from the JavaBean.

- *Customisation*. The developer tools supports customisation of the JavaBeans. The JavaBean enables this either through following the patterns that allows discovering the necessary information through introspection, and/or by providing additional support classes. Through a BeanInfo class a JavaBean can specify exactly the properties and events that should be visible in the tool, and also a special customiser class can be specified to allow tailored editing of the JavaBean.

- *Methods*. As the JavaBean is a standard Java class, any public method can be used by the developer.

In the original JavaBeans specification [Hamilton 1997] there was no standard for runtime containment (such as organising the beans in a hierarchical structure) or for beans to discover and use services provided by the environment. The BeanContext introduced in [Cable 1998] gives such a standard. The BeanContext interface allows adding and removing of children to the bean, and as it extends BeanContextChild it allows a hierarchical structure. The BeanContext can in addition support the extension BeanContextServices to organise and give access to a set of services. A set of events are also supported in the BeanContext standard to allow a listener to monitor changes in the containment hierarchy, and to be notified when services are made available or revoked.

Java 2 Enterprise Edition (J2EE)

J2EE [Sun 2003] is the standard platform for developing multi-tier enterprise application in Java. While the platform standard and a reference implementation are managed by Sun, implementations of the platform are available from different commercial providers, as well as from the open source community (JBoss).

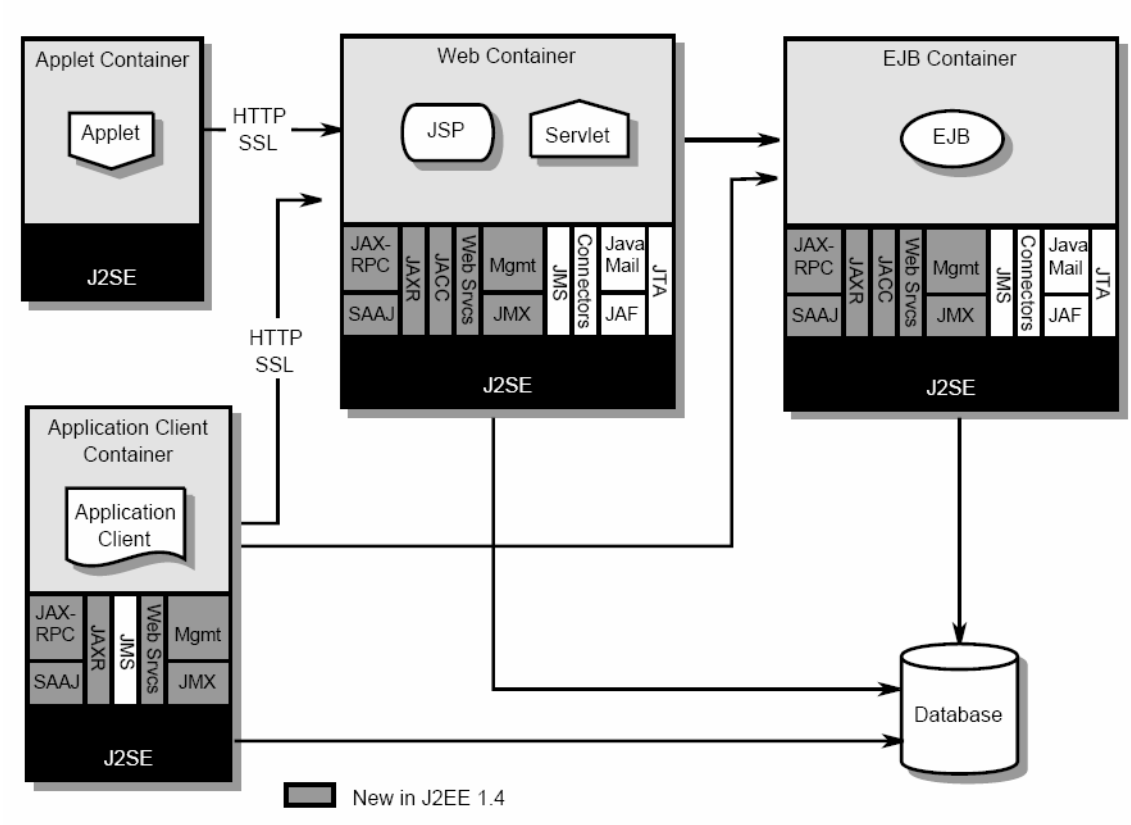

Figure 9 - J2EE Architecture diagram (from [Sun 2003])

As illustrated in Figure 9 J2EE defines four container types for application component. All the container types use J2SE as a foundation, and define a set of additional services supported by the container. Applet containers support running applets, and the typical example of such a container is a web browser. Application client containers provide a full execution environment for Java applications, typically with a richer user interface and without the "sandbox" limitations of applets.

Web containers support execution of different kinds of web components. Servlets components generate HTML pages based on HTTP requests. Built on top of the Servlets technology, Java Server Pages (JSP) provides a simplified approach to describing dynamic web pages. A special kind of Servlets is also used to support XML web services.

EJB containers support the Enterprise JavaBeans (EJB) component model, which extends the "write once, run anywhere" philosophy of Java to also include server side components. Enterprise beans typically contain the business logic of the application, and may also directly provide XML web services. The EJB container provides services such as transaction handling, security, remote clients, lifecycle and database pooling for the objects it handles, freeing the component developer from some of this work. EJB supports three different kinds of objects:

- *Session Beans*. These beans represent the client in the server, and perform tasks for a single client. They do not directly represent persistent entities, but can access and modify such entities. Session beans are not persistent; they are discarded when the client terminates.

- *Entity Beans*. These are persistent entities, each is typically stored in a row in a databases (can be any database, not just relational). Entity beans often represent business entities. Access to these beans is shared by multiple clients. Entity beans have a primary key that can be used by clients to retrieve the bean.

- *Message-driven objects*. These objects execute on receipt of a message, and are invoked asynchronously. The objects are stateless and do not represent persistent entities, but may access and update them.

3.7 Visual Component Environments

This section briefly describes visual environments that allow components to be organised into applications or documents. The theme of this section partly overlaps with end-user programming as presented in the previous chapter, but is presented here because it builds on the concepts presented in this chapter.

3.7.1 Visual Development Environments

User interface builders are so far the best example of visual, component-based development environments. From the early versions of Visual Basic and NeXT Interface builder, these tools have simplified the task of building user interface through visual composition of user interface object instances. One of reasons why these tools have been successful is probably that the domain of user interfaces naturally is well suited for visual composition. Many of the tools in addition also allow developers to create instances and edit the property values of non-visual objects as well, and the tools display these, e.g., in an off-screen area during editing.

Visual Age and Parts are two environments that take the visual editing a step further. Both have a history as Smalltalk development environments before Java versions were later developed. These environments allowed the developer to visually compose applications from components, and presenting visually instances of both user interface objects and other (non-visual) objects. In addition to setting property values and performing the editing usually done in user interface builders, these environments allowed connections to be made visually between the objects – both UI and non-visual. An example: to get the behaviour that pressing a button would add a string from a text input field to a (non-visual) list, a connection could first be drawn from the button object to a list and selecting the add method of the list to be triggered. Next, the developer would visually select that the parameter for the add method should be retrieved from the input field, selecting the getText() method from among the set of methods that returned values matching the parameter type. The connections created in this way could later be inspected and edited visually.

While this extended visual approach worked well in the tools, the main reason it has not become more widespread is probably that most programmers can write the same logic with just a few lines of code, and thus the tools (so far) do no significantly simplify the task. This is in contrast to traditional user interface builders which *do* seem to significantly simplify the task they support. Also, the user of the visual tool need to understand the method signatures provided by the objects, and to effectively use the tool (probably) need some programming experience. The tools are thus left without any real

target user group that prefer them. Hopefully, further development of tools in this category will overcome this problem, and one should more carefully select to support either professional programmers or a more pure component composer role.

3.7.2 OpenDoc

The most ambitious attempt to this date of introducing component-based composition environments to end-users was OpenDoc [Apple 1995][Curbow 1995]. The OpenDoc platform was conceived by Apple, and supported by Apple's partners IBM, Novell and WordPerfect. Not only a technology, but also a vision for the future direction of desktop software, it tried to introduce major changes both in user experience for document-based software, in the desktop software application market, and in software development technology.

From very early in the process, the OpenDoc platform design and development effort had a strong involvement from human interface designers [Curbow 1997]. The main idea of OpenDoc was to replace the dominating monolithic software applications with a component-based compound-document standard. Monolithic applications usually contain editors for many different kinds of content for a document, for example, text, pictures, tables etc. Often only one of the content types is the primary content type for the application. The result of this is that users often must learn many different editors for the same content type, and often must use editors that are not of their preference and which are suboptimal to the task. Further, the monolithic applications are often quite heavyweight, as they have to contain the editors of all these kinds of content.

In OpenDoc the focus is moved from applications to documents. A document consists of one or more parts, one of which is the root part of the document. Each part usually contains a single kind of content, e.g., text or picture. Container parts can in addition embed other parts within their content. The editor for handling a specific kind of part is called a part handler, and a part handler is distributed as a component. The user is allowed to select the part handler for a specific part from the components installed on the user's computer. This approach allows the user to combine different kinds of content in a document, and always use the preferred editor for each kind – independent on which vendor developed the editor.

One of the main features of OpenDoc user experience is the in-place-editing of different content, combined with an inside-out activation model. Briefly explained, in a composite document the user can edit the content of any part directly in its place within its enclosing part, and a single mouse click is enough to activate any part independent of its nesting level. When a part is activated, the menus of the document are adjusted to a combination of part-editor specific menus with a set of standard menus associated with the document. These standard menus support operations such as storage and printing of the document, and contain standard edit menu items such as clipboard operations.

From a market point of view OpenDoc was intended to challenge the position of the largest desktop application software vendors. The main idea was that by creating an open, compound document standard, it would be possible for smaller software companies to develop and sell high-quality specialised part handlers and in this way provide a viable alternative to the dominating monolithic applications. A standards body, the CI Labs (Component Integration Laboratories), was established to be the

owner of the technology, and was intended to promote the standard and help create a component-based software market.

Technically, OpenDoc can be seen as a Component Framework [Szyperski 1997]. The framework defines the interface which part handlers have to implement, as well as a set of rules for the part handler behaviour. Among the responsibilities of a part is to display and support editing of their content, and to store the content into the OpenDoc storage system called Bento. Rules also governed the use of shared resources, and part handlers had to follow a negotiation protocol to, for instance, gain keyboard and menu focus. The implementation of OpenDoc was based on IBM's System Object Model (SOM), which was a (partial) CORBA implementation. SOM also ensured that OpenDoc was programming language independent.

Although OpenDoc was a promising technology, it did not manage to reach the critical mass of acceptance by users, developers and the market. It was finally abandoned when Apple decided not to continue its development. The ideas of OpenDoc may have influenced parts of later designs and solutions, but no new platform incorporating a similar set of features and solutions have emerged. The reason why OpenDoc did not succeed is probably that it was too ambitious, and required too many changes in adopted practices at the same time. For the end users OpenDoc would become a viable alternative only when a set of part handlers were available which would allow them to replace and migrate from their existing applications. The part handlers that became available never reached the required maturity – maybe with one or two exceptions. Also, in their eagerness to rethink the user experience, the designers "fixed" some issues which the users considered as the standard UI rather than a problem, and this somewhat lowered the user acceptance (e.g., OpenDoc had no quit menu command, and renamed the File menu to Document). Existing software vendors hesitated to adopt the technology, both because they partially saw it as a threat to their existing applications, and because it was a fundamentally new way to create software that required a lot of developer training. One of the migration strategies for developers also failed, as the extension of existing monolithic applications to be OpenDoc containers proved to be quite hard.

3.8 User and Developer Roles

In the discussion of an industrial approach to software production [Reenskaug 1996] argues for defining a value chain of the people involved:

"We believe that the people who contribute their skills to the creation and deployment of software should be organized in a value chain. The guiding principle should be that while the qualifications of people on different layers will be different, the individual qualification requirements should be realistic in terms of a large and distributed organization plan. The professionals performing the tasks on each layer should be supported by a combination of technologies, procedures and tools."

Which roles are involved in the value chain of a software application can vary with the scale and type of application, with the size, type and culture of developer organisation, and with the application domain etc. For a typical development effort the following roles would at least be involved:

- *End user*. The user of the finished software application.

- *Application developer*. This developer uses an integrated development environment (IDE) to develop the application.

- *IDE developer*. Creates the integrated development environment, including programming language, libraries for platform, UI, DB etc.

[Reenskaug 1996] gives two examples of more refined value chains for large-scale production involving software. For intelligent network services the value chain contains the roles of end-user, subscriber, service provider, service creator, service constituent creator and network provider. For customised business information systems end-user, tool maker, module maker, kernel maker, system software supplier and production engineers are included. The details of these value chains are not repeated here, but the interested reader is instead referred to Chapter 10 of [Reenskaug 1996].

The most obvious role added by the introduction of component software is the component developer. This developer creates components that the application developer can use to simplify the development, or maybe even enable the application developer to create applications that would have been unachievable without the components. [Szyperski 1997] suggests that the following additional professions can emerge from component software development:

- *Component assembler*. Takes application requirements, selects components, and assembles the components into a working application. The work may range from largely automated by builder tools, to requiring extensive programming or scripting.

- *Component programmer*. Creates a component based on component specific requirements and specifications provided by the component framework.

- *Component framework architect*. Designs and implements a component framework that allows components to be plugged in and which interoperates with other frameworks.

- *Component system architect*. Analyses system requirements of existing and planned systems, and creates a system partitioning which enables a set of component frameworks to integrate while maintaining as much independence as possible.

3.9 Summary

This chapter has presented state-of-the-art in software reuse and component-based software development. Software reuse has long been recognised as central to achieve more cost efficient software development, but reuse has proven to be more difficult than expected and often organisational aspects are the most challenging. Although software components often are considered the main reuse technology, also other approaches such as application generators and very high level languages can be regarded as reuse techniques.

Software components were defined as (from [Szyperski 1997]):

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

Software components can be reused both using black-box and white-box reuse, although white-box reuse using subclassing can cause problems when new versions of the component containing the subclass are released. Adaptations may be required to reuse as a component when it is not possible to reuse the component "as-is".

Software architecture defines the fundamental organisation of a system, and defines the roles of central components, their collaboration, and how to interact with shared resources. Component frameworks partially enforce architectural principles, and increases the chance that independently developed components can be used without problems in a system.

This chapter has also briefly presented the current mainstream object and component models, including CORBA, COM, .NET, Java/JavaBeans and Java 2 Enterprise Edition. A brief look was taken on visual development environments, and OpenDoc – an (now abandoned) component framework for compound documents. The chapter closed with a discussion of developer and user roles associated with component-based software development and industrial production of software.

Chapter 4 Case Studies and Example Introduction

This chapter presents two case studies and introduces an example. The case studies are based on research done in European research projects we have been involved in, where tools for end-user programming and composition of software were developed. The domain of these cases is personalised tools for disabled and elderly users. Each case is presented with a brief presentation of the project followed by a general and a technical description of the system, and concluding with a summary of experiences from the case. A summary of the two cases and some of the experiences has also been published in [Stav 2000]. After both cases have been presented, a short section compares the two cases.

The two cases are typical examples of the kind of environments for which this thesis suggest development methods and tools. The cases have provided us with good background experience on development of such environments, and this experience has been of great value when developing the methods and tools proposed later in this work. The two case studies contribute to this thesis with:

- requirements for similar systems;
- experience with the software architectures used in the systems;
- reusable components developed in the project and/or knowledge about components that would have been useful to the project;
- mechanisms that were found to be useful or required in such systems;
- experience on what was time-consuming in the implementation;
- problem areas that remained unsolved or only partially solved in the projects;
- other knowledge on what worked well and what did not work so well.

The example introduced here is part a role-playing computer game generator, with focus on the story-telling aspect. The purpose of this example is to present a realistic case on which the proposed meta-toolkit for end-user programming environments can be tested. It differs from the case studies both by being from a different domain, and by not being an existing system. Instead it is an example of a new system that could be built using

the proposed methods and tools, and is used throughout the description of the proposed frameworks and tools to illustrate how these can be applied. This example was selected due to personal interest and experience with the domain, and was defined as part of the thesis work.

The choice of a set of cases and application domains may influence the design and limit the generality of a proposed solution. This risk is greatly outweighed by their contribution to requirements and solutions, and by being concrete cases to test more abstract ideas on. A greater number of cases and domains would reduce the risks, but the selection made here was made to balance this with the required effort and time.

4.1 Comspec

The Comspec project ([Svanæs 1993], [Lundälv 1998], [Head 1998]) was a European Union funded research project within the TIDE programme from 1994 to 1998, and with an earlier history founded by the Nordic Council. The project had participants from six European countries, and included participants from research, rehabilitation industry and rehabilitation institutions. The objective was to develop an extendable software environment allowing personal communication aids for disabled users to be created without traditional programming.

4.1.1 General Description of Case

The end-user group of the software developed in the Comspec project was disabled users who needed tools for Augmentative and Alternative Communication (AAC) and Alternative Access (AA). An example application could be a symbol-based (for instance, Blizz) aid with synthetic speech output, where the end-user navigates and selects what to say from a set of displayed symbols using one or two switches. The idea of the project was to develop a modular software environment for creation of such personal communication aids, where modules from different developers in the rehabilitation industry could be integrated.

Five different user roles were identified in the Comspec project [Svanæs 1993]:

- *End-user*. The disabled user of the final communication aid.
- *Facilitator*. Someone close to the end-user (e.g., a teacher, therapist or parent) that can do day-to-day minor adjustments to the aid like adjusting scanning speed.
- *Integrator*. A person with some more technical knowledge that is able to compose a personal aid for the end-user using Comspec tools and components.
- *Developer*. Someone that can make a new component that will make some new features available from the Comspec environment.
- *Platform Developer*. The developers responsible for developing and maintaining the tools and frameworks of the Comspec environment.

In the TIDE phase of the Comspec project, OpenDoc ([Apple 1995]) was originally selected as the component technology to be used. This greatly influenced the design and architecture of the early development of the software. The document centric nature of OpenDoc was well suited for most of the editing and storage functionality of Comspec, and many of the UI design questions where resolved using the guidelines of OpenDoc. When the creators of OpenDoc decided to abandon the technology, a decision was made in the project to move the development to Java/JavaBeans. Along with this decision, most of the resources planned for the evaluation phase were redirected to the Java/JavaBeans based reimplementation effort, thus scaling down the evaluation phase to enable the development of a usable demonstrator.

The Java/JavaBeans-based ComLink software that resulted from the Comspec project consisted of two applications – an editing environment and a runtime application. With the editing environment a personal communication aid for a disabled user can be created. The runtime application is used to run the aid for the disabled end user.

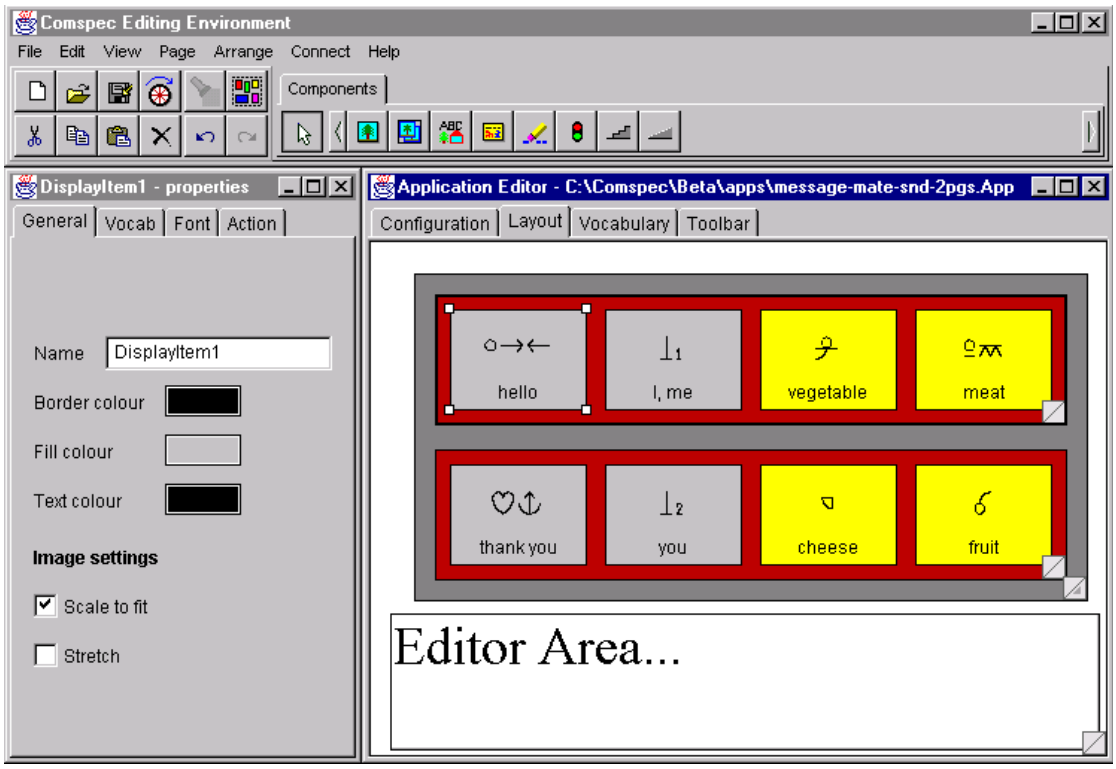

Figure 10 - The layout editor of the ComLink editing environment

A ComLink aid contains a set of subsystems. In the version that was available at the end of the Comspec project the software contained four subsystems: layout, configuration, vocabulary and toolbar. The *layout* defines mainly what the end user will see on the screen. Figure 10 shows the ComLink editing environment with the layout editor opened. The palette of the top window always displays object types available in the current subsystem, while the property sheet at the right shows the properties or a customiser for the selected object. From the *configuration* the flow of events and output from the system is defined. Figure 11 shows the configuration editor of the ComLink editing environment. Connections can be made between two objects in the configuration when they have ports matching the sender and receiver roles of the same protocol. The terms the user uses in his/her communication are defined in the *vocabulary*, and for

each of these terms there may be one or more representations (picture, sound, text, bliss symbol etc.). From the *toolbar* it is possible to define a visual dialog that enables adjustment of some parameters of the aid, e.g., the speed of the scanning cursor used when operating the system through a single switch.

All of the existing subsystems can be extended with components introducing new object types:

- *Layout elements* are visual elements, usually viewers/editors or containers used to compose the layout. Most layout elements have associated customisers that make editing of properties easy for the facilitator and the integrator;
- *Configuration elements* represent various input devices, selection methods, outputs and event flow controllers, and have ports allowing connections to be defined to ports of other configuration elements;
- *Vocabulary representation types* are used to associate various media types with vocabulary items, and are usually either displayable in layout elements through an associated viewer, or used by configuration output elements (e.g., text-to-speech);
- *Toolbar elements* are visual controls that allow adjustment of parameter (usually a specific type; e.g., integers, boolean) by the facilitator from the runtime system.

The ComLink environment is also extendable with entire new subsystems; the Toolbar subsystem was, e.g., added late in the development.

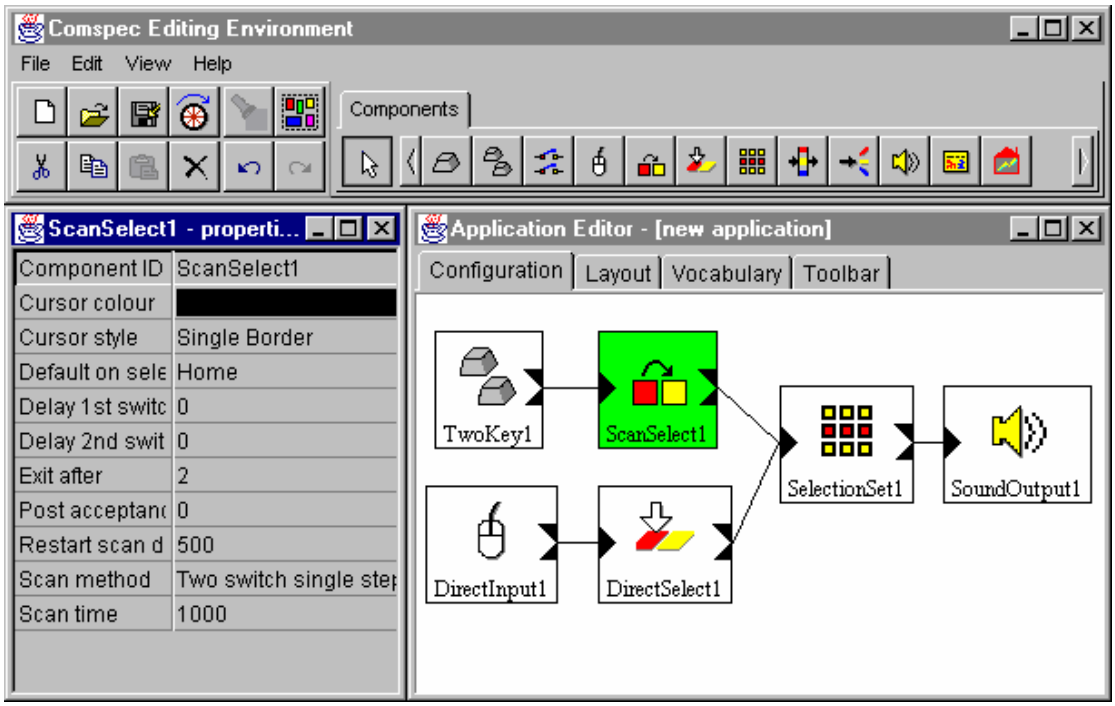

Figure 11 - The configuration editor of the ComLink editing environment

In the OpenDoc version of Comspec some design questions were answered directly by the human interface guidelines of OpenDoc. When the software was reimplemented in Java these questions had to be addressed again. The layout editor was the most affected in this respect, as it allows hierarchical nesting of layout elements. After some discussion a slightly modified version of the inside-out activation model ([Curbow 1997]) of OpenDoc was selected. A single click causes the deepest nested container at the cursor position to be activated, and if the click is at a non-container layout element, this element is selected within the container. As each container can have an associated editor, it is the responsibility of the editor to implement these rules. An editor can also give access to a popup menu for actions associated with the container. The reason for having a special editor for each layout container is that containers may have different restrictions on placement and addition of other elements within them. Also special commands may be applicable to a container. The editor of the layout stack element, e.g., has a popup menu that allows adding and removing of cards in the stack.

4.1.2 Technical Description

Architecture

In the initial OpenDoc development of Comspec the architecture was greatly influenced by the technology. The system was envisioned as a document-based system, with OpenDoc part handlers as the components of the system.

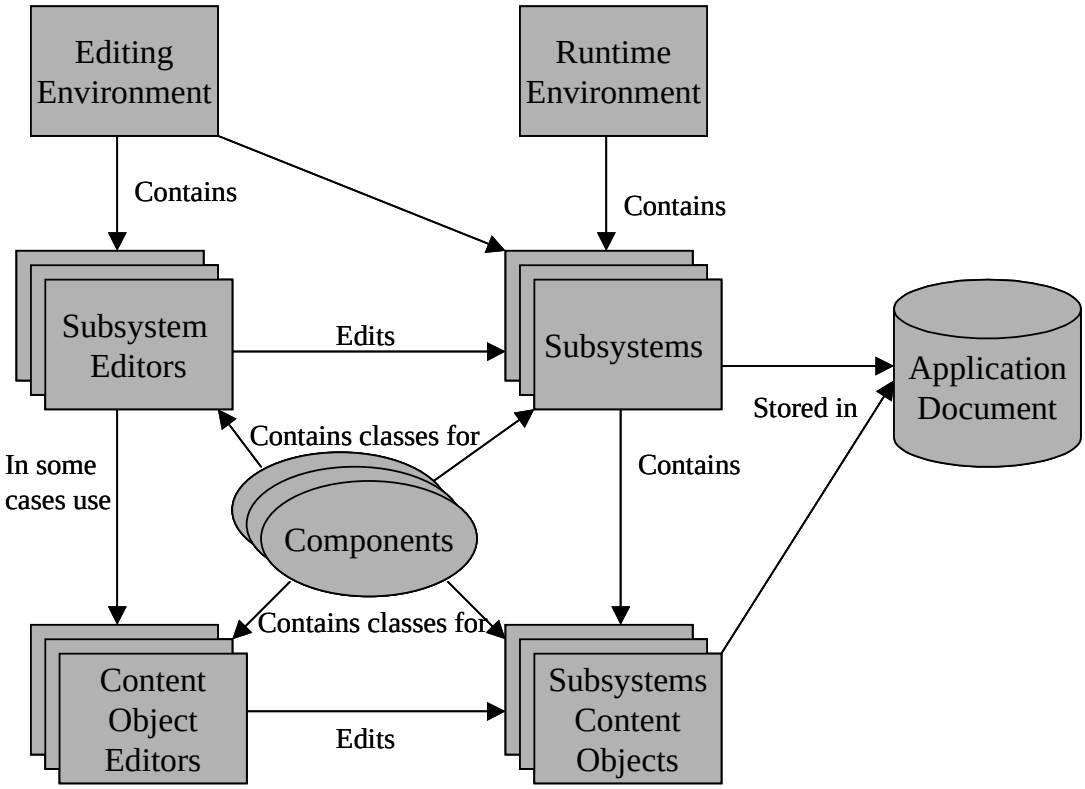

Figure 12 - Overview of ComLink software architecture

When the development was moved to Java the architecture was rethought, and JavaBeans was selected as the component technology. The document remained a central

part of the system, as it contains the persistent version of the application. As displayed in the overview in Figure 12, two applications were developed: an editing environment and a runtime environment. Further, the application was divided into subsystems, as already mentioned in the general description of the system.

Each subsystems usually has an extendable set of content objects types, e.g., layout and configuration elements. For each subsystem there is an editor specifically made for it. Some of the subsystems editors can also be extended with editors for specific content object types. Following the JavaBeans standard components are contained in Jar files, where each Jar file contains all the classes for one or more components (for example, new elements with belonging editors, customisers etc).

The runtime architecture of the ComLink software was designed as a set of role models [Reenskaug 1996], and from these a set of standard runtime protocols were developed. Figure 13 displays a UML model of the main object roles involved in these protocols (the details of each protocol are omitted here). The *control protocol* is used for interaction between an input object (e.g. switches, keyboard, mouse) and a selection method (e.g. mouse select, one/two-switch scanning). The *navigation protocol* is used between a selection method and the set of possible selections it navigates through. The selection set uses the *layout protocol* to communicate with and find links between individual layout elements, while the *vocabulary protocol* is used to send vocabulary items of selected layout elements to outputs.

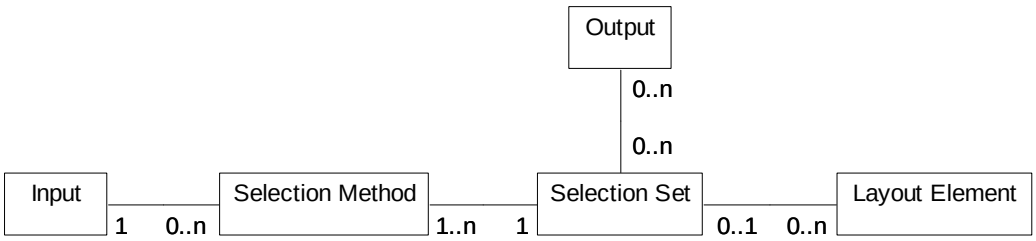

Figure 13 – UML Diagram of the object roles in the ComLink runtime protocols

In the initial OpenDoc development, the storage and editing architecture were most influenced and directly supported by the selected technology. When the software was moved to Java/JavaBeans replacements for these had to be found.

The Java serialisation mechanism was investigated as a candidate for a persistent storage solution, but was found to not give the flexibility required. An illustration of the problem was the incompatibility between different versions of the serialisation of Swing UI objects, from which some of our components were developed. Instead a new storage mechanisms was developed from scratch, that in principle allowed components to evolve, and also was readable even if some of the components used during storage was missing in the reading environment.

As already mentioned, the layout editor was most affected by the move from OpenDoc to Java, as much of the original design was derived from the OpenDoc UI guidelines. In the selected solution each object type used in the layout are allowed to have an associated editor type.

Components

Components can introduce different kinds of extensions to the ComLink system. Each of the existing subsystems in ComLink can be extended with new object types. A ComLink component consists of one or more Java classes with associated resources, usually packaged in a Jar file. The ComLink software uses a class registry to find which object types and subsystems are available, and which classes implement the various services it depends on. During runtime, the class registry is held by the editing environment (see Figure 14), and contains an entry for each object type. This persistent representation of the registry is maintained in a text file, and to make the system aware of new object types introduced by a component, an entry for each object type has to be added to the class registry file. In the Comspec project it was envisioned that the necessary updates to this file would eventually be made by a component install tool, but due to limited resources such a tool was not implemented in the project.

New object types usually support some of the runtime protocols based on the roles they have at runtime in the system. They also have to support some general mechanisms both at runtime and edit time. All object types introduced by ComLink components must support a protocol that gives the context for the object instances (similar to the BeanContext mechanism available from Java v1.2). The context (ComspecContext in Figure 14) is used by the instances to locate services.

As is shown in Figure 14, the editing environment maintains editors for zero or more applications. Each application contains a set of subsystems, and correspondingly the application editor contains subsystem editors for these. Subsystems and subsystem editors are coarse grained components, allowing ComLink to be extended with new subsystems both at runtime and in the editing environment. Through the use of ComspecContext and ComspecContextChild the editors are organised in a hierarchy, and this hierarchy can be used by the editors to locate required services.

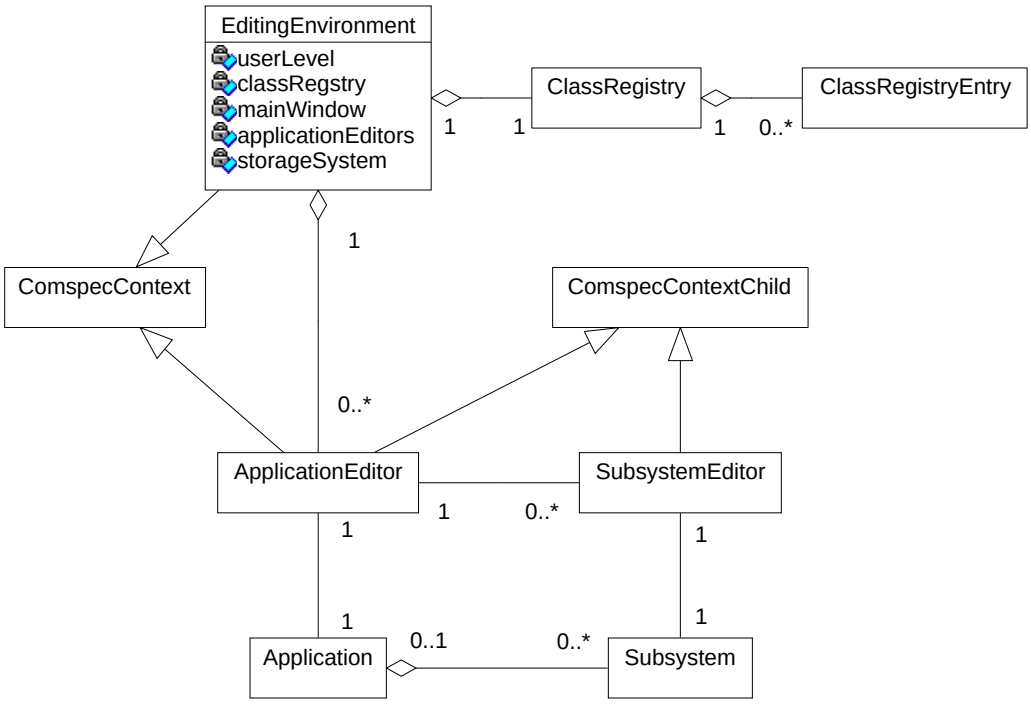

Figure 14 - Central classes of the ComLink editing environment

In the layout subsystem it is possible to introduce new visual object types. Instances of these types will usually be visible to the end user on the screen. These object types must support a protocol that handles layout specific information (in addition to the common protocol described above). Following the JavaBeans model, each object type can have its own customiser for editing of properties that is displayed instead of the property list. Also, layout elements, and containers in particular, can have associated custom editors that is nested in the layout editor, and used for, e.g., selection and arrangement of sub-elements. These editors are required to implement some protocols (somewhat inspired by OpenDoc, but much simpler) to co-operate suitably in the editing process.

The configuration subsystem can also be extended by components. New objects types in the configuration will be visible to the integrator, and allow extension of the flow and handling of events. These types must support the configuration protocol that is used to describe what ports instances will have, and to manage connections to the ports. In addition each object type will usually support one or more signal types that the instances can send and/or receive.

The vocabulary subsystem can be extended by components to introduce new types of representation; e.g., video, speech, etc. Representation types will usually implement some of the associated protocols. A representation viewer interface is implemented to allow viewing in the layout, while a representation editor interface is implemented to support configuration of the representation for a vocabulary item.

It is also possible to introduce new subsystems in ComLink, or to replace the implementation of one or more of the existing subsystems. Components introducing new subsystems will usually be more coarse-grained than components extending a subsystem.

Additional mechanisms that may affect components:

- *Interface-based object retrieval*. Some objects need a way to find other objects that are candidates for having a collaborating role. Selection among such candidates may, for example, be done in a list presented by a property editor or customiser. In ComLink this is realised by an object registry where objects can be retrieved based on a set of interfaces the object should support.

- *Context-unique name*. To enable selections like the one mentioned in the previous bullet-point, each selectable object needs a name that is unique (at least in a suitable context). To ensure unique and legal naming, objects are required to register their name with a central object registry. The registry will not allow two objects with the same name, and it also checks other naming rules. If registration fails, the object that attempted to register has to select another name or revert to its earlier name.

- *User level notifications*. Editors that have behaviour depending on the user level can register to get notifications when the user level changes. When notifications are received, their behaviour and appearance can be adjusted as appropriate.

- *Document state notifications*. Some of the allowed and wanted behaviour of the objects depended on the current state of the document they are located in. An example is that some service objects require all objects to have finished their

initialisation before they are usable, and some other objects wants to hide some editing widgets when the tool is switching from an editing state to a testing/running state. Such objects can register for notifications when the document state is changing.

4.1.3 Experience

Along with the design of the architecture and main mechanisms, the most time-consuming work in developing the ComLink software was the design and implementation of the subsystem editors, and in particular the layout editor. The layout editor is a hierarchical visual editor, not unlike what is now present in most user interface builders. The main special feature that was required in the layout editor was support for visually displaying navigation links among the visual elements. In addition to the design considerations already described for it, usability issues like undo-support contributed to the amount of required work. An extendable visual editor component or a framework for building visual editors would have been useful for the project, but at the time of implementation no suitable candidates (Java AWT or Swing-based) were found.

The other subsystem editors were less demanding than the layout editor, but also these would probably require less effort to implement if suitable components or frameworks had been available. The configuration editor is in principle a general connection-based editor, and something quite similar could, e.g., be useful for editing connections in a component model like CORBA Components. The vocabulary editor is a browser-like editor.

Within each of the editors a palette of object types are used to create new objects, and from each of the editors a property/customiser panel is used to configure the objects. This corresponds quite well to the general JavaBeans model. A property panel was implemented for the system, as no suitable component was found. These kinds of tools are currently included in most development environments, but are usually not reusable components available for redistribution. Customiser panels generally give a better UI for editing the properties of objects, but require some work to code for each object type. This could be an area where tool support could reduce the development effort, by providing a visual tool for creating the panels and using reflection or code generation to reduce the required handwritten code.

Persistent document-based storage is another area where the implementation of the ComLink software could have been simplified had a suitable solution existed. Among the requirements found in this area that will probably turn up in similar systems are:

- handling of references to other object, including both strong and weak references;
- different properties to be persistent at edit- and runtime.

Many of the mechanisms that were implemented in the ComLink software will be usable in similar environments. These include the mechanisms already described for class registry, interface-based object retrieval, context-unique naming, user level notification and document state notifications. Most of these mechanisms do not require much implementation work, but are reusable ideas not unlike like software patterns

[Gamma 1994]. A context mechanism similar to BeanContext is a requirement for all of these services.

The definition of a set of user roles helped to guide the development by focusing on the needs of the various groups, and to early build in support for different user roles in the software.

The division of the software into separate subsystems was, at least with the limited experience available, successful both from a tool user and a more technical viewpoint. As each of the subsystems have tools and sets of object types suitable for their specific purpose, the application developers (integrators / facilitators) are guided in their work. Each subsystem presents a manageable and understandable part of the job, and chances for making mistakes or meaningless designs are reduced. The framework of each subsystem and predefined protocols allow meaningful extensions to be made that will cooperate with the system both at edit- and runtime.

The ComLink editors support both the integrator and facilitator roles. The facilitator will not see all of the subsystems, and some of the editors offer only limited functionality for this user group. In the layout editor a facilitator is only allowed to modify the properties of existing object instances, and cannot move, resize, add or remove objects. This approach works fine, but there are some issues that still are a bit problematic. The facilitator does not see the configuration at all, and thus cannot directly change anything related to the configuration. Though the configuration as such may be too difficult to handle for a facilitator, some properties connected to the configuration, e.g., speed of automatic scan, can be important for the facilitator. These properties are associated with objects in the configuration that the facilitator does not even know that exist. So how can these properties be presented to and edited by the facilitator? The solution selected in ComLink was to allow these kinds of properties to be available in the toolbar. The responsibility for selecting the important properties of various objects and making them available on the toolbar is laid on the integrator.

A goal in ComLink was to have independent extensibility, but this goal was only partially achieved. A new component in the configuration should, for example, be able to introduce a new style of navigation among the elements of the layout. The existing navigation components base their navigation on a predefined set of links that each object in the layout keeps track of, and that can be edited in the layout editor. New components can without problem be introduced in the configuration, but the kind of navigation allowed is restricted by the predefined properties of the layout objects, including the set of predefined links. If the navigation method logically is based on another set of links, it has no way of extending the layout objects to keep track of these, and also no way of extending the layout editor to allow editing of the new set of links.

Could another design fix the problem? It is of course easier to find a general solution to a problem once you know what the problem is with a specific implementation. In the case of the navigation links a possible solution would be to have an extendable set of links between objects in the layout, with a more general handling in the layout objects and in the layout editor. In the more general case during design a balance must be found between being too static and specific to the problem at hand (preventing extensibility), and being too general (leading to over-designed systems).

Some suggested additions to object orientation could to some degree address the problem. In subject oriented programming [Harrison 1993] an object can be extended to allow different "views" on it. The technique described in [Harrison 1993] is based on recompilation of the source code of the involved objects though, and would not be directly suitable in a component-based environment.

As the evaluation phase of the Comspec project was scaled down due to the change of technical platform, the primary evaluation was limited to two workshops for rehabilitation professionals filling the facilitator and integrator roles. The main experience from these workshops was however that the participants soon were able to understand and create ComLink applications [Head 1998].

A summary of the main technical experience from the Comspec project is presented in Table 1.

Table 1- Summary of experience from Comspec

Category	**Summary of lessons learned**
Most time-consuming activities	Design of architecture and main mechanisms Design and implementation of subsystem editors
Main mechanisms used which could be reusable in other systems	Document-based storage Hierarchical context (for finding services) Class registry Object registry (with interface-based object retrieval) Context-unique naming Document state notification User level notification
Reusable parts and tools that could have reduced the required effort and simplified development	Hierarchical visual editor (e.g. layout editor) Connection-based editor (e.g. config) Browser-like editor (e.g. vocabulary) Property panel (generic editor for properties) Tool for making customisers (editors for properties) Extendable visual editor component or framework, which could be the used to create the hierarchical visual editor and connection-based editor

Other useful experience	Definition of a set of user roles Division into subsystems, with extensibility within each subsystem
Issues that remained unsolved in the project	How to appropriately hide subsystems for some user groups, while showing information from the subsystem which that group requires Limits to independent extensibility (e.g. navigation links)

4.2 TASC

TASC (Telematics Application Supporting Cognition) ([TASC 1999][TASC 2000]) was a European Union funded research project in the Disabled and Elderly Sector under the Telematics Applications Programme within the Fourth Framework Programme, running between 1997-2000. The project included participants from seven European countries, and included participants from research, software industry and health care providers.

4.2.1 General Description of Case

The objective of the TASC project was to develop tools to support decision-making, planning and communication for persons with cognitive disabilities. The TASC software included support for planning and prompting of tasks which the user has to remember throughout the day, simple and personalised access to information sources and communication abilities, and supported picture- and symbol-based communication for people with limited reading abilities. The TASC software was developed in Java.

In relation to the TASC software four different user roles can be identified (although these were only implicitly defined in the project):

- *End-user*. A person who is in need of cognitive support for tasks like planning, decision-making and communication.

- *Carer*. Someone close to the end-user (e.g., a teacher, therapist or relative) that can set up a user profile for the end-user tool by selecting relevant components, and setting up some predefined content like address books and plan templates.

- *Developer*. Someone that can make new components that will make new features available in the TASC environment.

- *Platform Developer*. The developers responsible for developing and maintaining the frameworks and applications of the TASC environment.

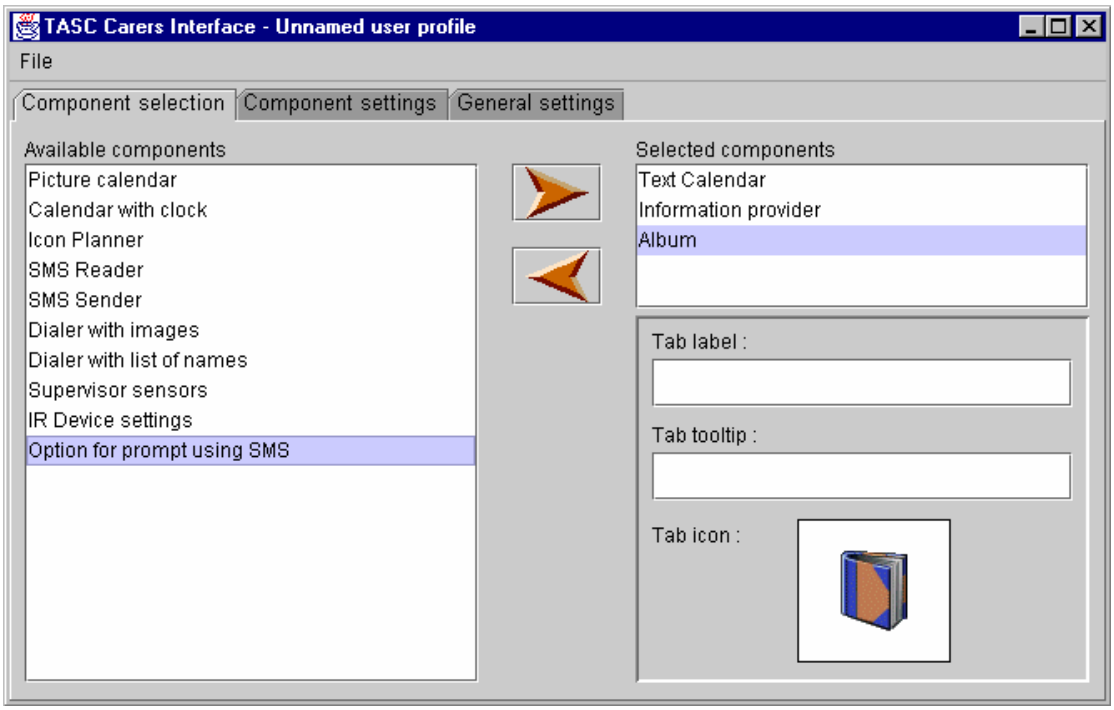

Figure 15 - The TASC carer's interface

Similarly to the ComLink software, the TASC software consists of a builder and a runtime application. The carer uses the builder application to select and configure a set of tools for the end user. Figure 15 shows the builder application, displaying the editor where the carer selects which components to include in the user profile of the end user. Settings and content for each of the components are edited with component-specific editors in the "Component settings" view. The end user uses a runtime application that contains a tab sheet with a tab for each tool which the carer included in the user profile. The set of tools available is extendable.

The granularity of the components used in TASC is larger than the usual components in ComLink, and is as such more comparable to the ComLink subsystems.

Among the tools developed during the TASC project were:

- Text-, icon- and symbol-based time planners that allow the user to plan the day. The carer may set up predefined task templates that the user easily can select from. Reminders can be presented on the PC or be sent as SMS messages to a cell-phone.

- Picture- and text-based phone diallers that help users select persons to call and recognise who is calling.

- Environment sensors that warn the user of situations like an overheating oven, open refrigerator door, fire alarms etc.

- Information provider that presents updated information of importance to the user in a format the user can understand; e.g., local weather forecast, bus tables etc.

4.2.2 Technical Description

Architecture

TASC end-user applications are built around user profiles that, similar to in ComLink, are stored in documents. Figure 16 shows an overview of the TASC architecture (from the technical documentation of the TASC software [TASC 1999]). Through the builder application the tools to be included in the end-user application and their initial content are selected and organised in the user profile. The user profile is stored in a file that is read when the runtime application starts. The runtime application stores the profile at regular intervals and at application exit, thus making changes to the content persistent.

Seen from a developer's viewpoint the TASC architecture contains more than the users see. The tools seen by the users belong to a user interface layer, while much of the behaviour is handled by entity- and service-objects in the "business" layer. This allows multiple views of the same information, and is used, for instance, to have three different planner tools adjusted for different user groups based on the same plan entity. Collaboration that appears to occur directly between the tools also actually happens at the business layer, with the tools being updated through notifications. Figure 17 shows the layers in the TASC architecture (from the technical documentation of the TASC software [TASC 1999]).

For the purpose of the TASC project the Java serialisation mechanism was found to fill the needs for a persistent storage solution. As part of further development to a product another solution may be required for some of the same reasons as mentioned regarding the ComLink storage mechanism.

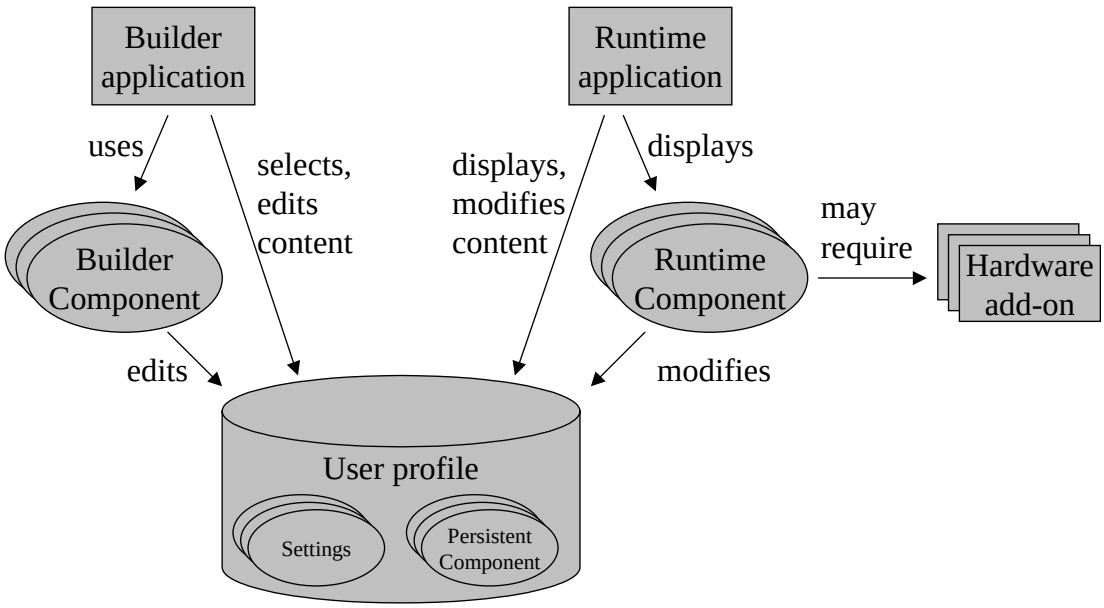

Figure 16 - Overview of TASC software architecture.

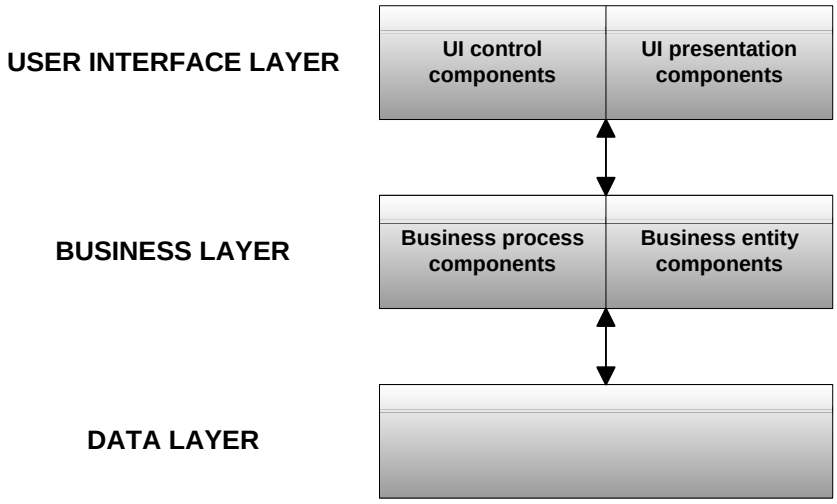

Figure 17 - Layers in the TASC software architecture

Components

A developer can extend the TASC software with new components. Usually an extension will include a tool seen by the end user, and a corresponding editor used by the carer to configure and create content to be displayed in the tool. It may include new entity or service objects used by the tool, or just give a new tool on top of existing "business" objects. TASC uses the same class registry mechanism as ComLink to describe which components are available to the system.

The component types supported by the TASC software are (see also Figure 18):

- *Persistent (business entity) components* implementing the PersistentComponent interface, e.g., the Plan component used by all the time planners.

- *Service (business process) components* implementing the TascComponent interface, e.g., the Scheduler used to prompt due tasks from the plan.

- *Runtime (UI control) components* implementing the RuntimeComponent interface are the tools seen directly by the end-user, e.g., text- and icon-based time planners.

- *Builder (UI control) components* implementing the BuilderComponent interface are the tools used by the carer to create settings for runtime components and to edit content for objects crated from the persistent components.

The TASC environment is not separately extendable by UI presentation components, but instead this kind of components, such as the Swing components of Java, is used to build Runtime and Builder components.

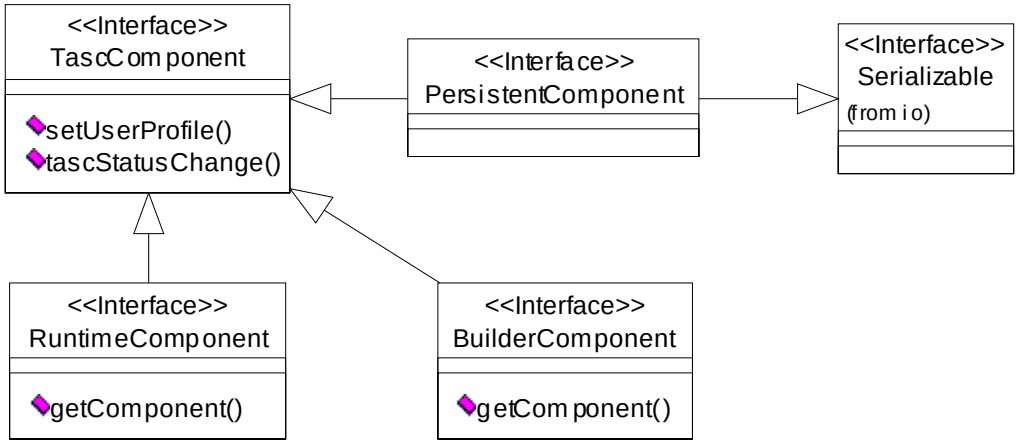

Figure 18 - Component types in the TASC system

All components of the TASC system relate to the rest of the system through the user profile (see Figure 19). Whenever an instance is created from a component, it is notified of the user profile it belongs to. Through the user profile instances of services and persistent components can be found based on their interfaces. All components are also notified of changes to main state of the editing environment, e.g., start and stop of runtime operation. This is, for instance, used to start and stop the prompting of the scheduler.

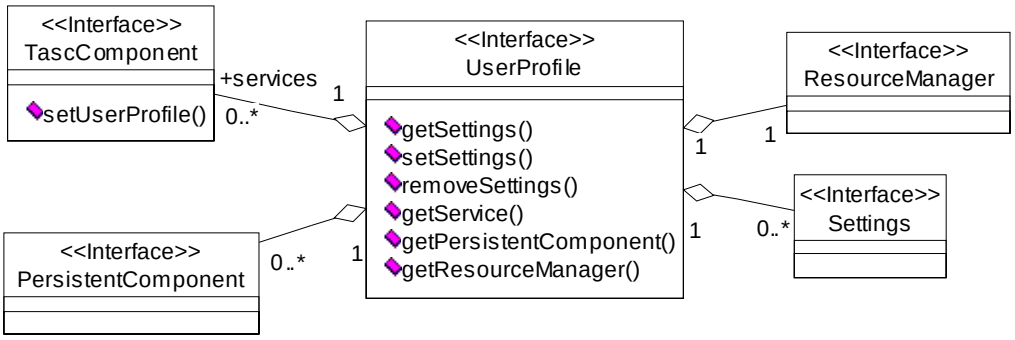

Figure 19 - The TASC user profile and related objects

4.2.3 Experience

Most component types in TASC are coarse grained compared to the components of ComLink. This simplifies the job of selecting and adding components to include in a user profile. As typical components of the TASC system were fairly independent of each other, it was easy to get the application as a whole to work. The simple access to services and other components through the user profile was found to be suitable for the coordination needs of the components.

In the TASC project the main focus of the UI design work was on the end-user application. The end-user UI was critical to get acceptance of TASC software the primary users. The evaluation indicates that this design and the following implementation effort were successful in this regard, at least for some of the main

components [TASC 2000], although instability of early versions of some of the components was a challenge to user acceptance.

The Builder application was added late in the project when it became clear that such a tool was needed, and less time and resources was available for its design and implementation. A simple approach was selected, in which a list of available components are presented to the carer, who then selects which components to include in the user profile. For each (main class of) component there is a single instance. For each selected component there can be an associated builder component. This editor becomes available when a component has been included in the user profile. This approach was selected for its simplicity, but is probably not an ideal solution. With the selected solution, the carer is not made explicitly aware of the business level components. Thus the carer may have a mental model that only includes the end-user components, with associated editors. This model will not match certain cases that can be experienced. As already mentioned there are multiple end-user components that display the planning part of the system. It is, for example, allowed by the builder tool to include both a text- and symbol-based time planner in the system, and this can be useful to allow the carer to help with planning while the end-user application is running. In these cases it becomes apparent that the user interfaces works on the same business entities. Also, the builder application will present two almost identical editors on the same data.

Reports from testing of the TASC tools indicated that some of the carers had difficulties with editing of settings for components from the builder tools [TASC 2000]. The number of available options seemed to overwhelm some of the carers. Part of the reason for this was probably that the TASC Builder UI is not consistent across components as the customisers for the components were developed separately and without any specific design rules or common style. Creating these editor components was also a somewhat time-consuming task. This is an area where tool support could have reduced the effort, and provided more consistent user interface across the components.

Another weakness with the software available at the end of the TASC project was the lack of management of the business objects by the builder tool. The “persistent components” and “service components” to be used were defined directly in the class registry text file. This file usually included entries for all business objects available for all components, and (empty) instances of all these objects were included in all user profiles, even if they were not used by any of the UI components. The overhead with this approach was probably not very large, as the total number of business object types was limited, and no practical problems due to this were experienced.

Time and resources did not allow exploration of other approaches as part of the TASC project, but one idea that emerged was to make the business-objects more visible to the carer. When the user picked UI components to include in the user profile, the business-objects these components required could be automatically included, and displayed to the carer. Editors for predefined content could then be associated with these business-objects instead of with the end-user UI components. Settings for the end-user UI components (e.g., selection UI-component specific colours, fonts etc) could still be edited by builder components associated directly with the end-user components.

Table 2 shows a summary of experience from the TASC project.

Table 2- Summary of experience from TASC

Category	Summary of experience
Most time-consuming activities	Design of architecture and main mechanisms Design and implementation of runtime components (anticipated in project plan) Design and implementation of editor components (not anticipated in project plan)
Main mechanisms used which could be reusable in other systems	Class registry Document state notification User profile
Reusable parts and tools that could have reduced the required effort and simplified development	Tool for making customisers (editors for settings and predefined content)
Other useful experience	Coarse grained components worked well for users and developers
Issues that remained unsolved in the project	Usability problems with Builder Builder management of business objects

4.3 Discussion and Comparison of the Cases

When comparing the roles identified in the Comspec and TASC case, the end-user, developer and platform developer roles are similar for both the cases. In TASC a single carer role was identified, while in Comspec this was divided in a facilitator and an integrator role. TASC's carer role has to handle all the preparation of predefined content, and also selection of and settings for the coarse grained components to include in a user profile. Compared to Comspec this includes the responsibilities of the facilitator as well as parts of the integrator's work. TASC does not require the same amount of definition of behaviour as is, e.g., done through the ComLink configuration editor, and thus there is no parallel for these parts of an integrator's responsibilities in TASC.

When comparing the environments provided to the carers, some elements appear in both cases. The most obvious of these is that both environments provide content editors for a variety of object types. In ComLink the structural part of the editing is mostly handled by the subsystem editors, while the details of each object are edited through customisers provided for each content type, or trough the property editor if no customiser exists. In TASC, both the structuring and the details are handled by the coarser grained builder

components. Building these editors is a somewhat time-consuming and repetitive task in both cases, and it would probably benefit both the developers and the users of the editors if some high-level tool could partly automate and provide a common style for this work.

The TASC architecture illustrated in Figure 16 and Figure 17 can at first sight seem to have a better separation between user interface and business layers than ComLink as illustrated in Figure 12. However, a closer look at ComLink reveals that it also have a good separation between these layers. The subsystems and subsystem content objects constitute the business layer, while the subsystem editors and content object editors constitute a user interface layer. At runtime the separation is not as clear as during editing, and this is partly due to the nature of software created in ComLink where most of the content edited by the carer directly constitutes the user interface of the end-user.

As the components in TASC are usually coarser grained than most ComLink components, they are in complexity somewhere between the ComLink subsystems and subsystem content objects. A difference between the architectures is that a significant part of the extensibility of ComLink comes from the fact that each subsystem is a component framework defining extensibility of its own (e.g., allowing new layout and configuration object types). In the TASC architecture no extensibility mechanism were defined for use within the coarse grained components. For a further extension of the planning and prompting components of TASC it would however probably be a good idea to create a component framework which would allow extensions to be made for task-specific support, e.g., procedures integrated with “smart” equipment for safe medication or safe cooking for people with memory problems.

Document-based storage was selected as persistence mechanism in both of the cases. The mechanism used in ComLink was more robust than the Java serialisation mechanism used in TASC, but required more code to be written for storage handling. Serialisation was selected in TASC mainly due to limited implementation resources, and a more robust solution would be preferable if resources allowed it.

In both cases a mechanism supported localisation of services provided in the system, but while TASC only had a centralised location for finding services in the user profile, ComLink used the context hierarchy for getting access to services and thus allowed more localised services, e.g., at subsystem level. In both cases there was a mechanism for reacting to system state changes, although the TASC variant was somewhat simplified compared to the ComLink document state notifications. As TASC did not have different carer roles, there was no parallel to the user level notifications of ComLink.

A parallel to the ComLink mechanisms for context unique naming of objects and interface-based retrieval of objects was not needed in the TASC system. In ComLink these mechanism was used to find and edit connections between objects from different parts of the system, and would also be of use in an intended scripting extension of the system. This level of fine grained integration was not required in TASC, but the need might have emerged if the coarse grained components were extended to be component frameworks supporting more fine grained components.

4.4 Role-Playing Game Construction Environment

The purpose of the example introduced in this section is to show a realistic case which can be used to illustrate the frameworks and tools proposed later in this thesis. While the case studies describe existing systems, the partial design of the example presented here is not part of an existing system. Implementations of different parts of this example could be used as proof-of-concept for a future implementation of the frameworks and tools proposed in this thesis.

Role-playing games is a popular genre of computer games. Games in popular series like Final Fantasy[12] and Baldur's Gate[13] has been sold in millions of copies, and is created in large productions often involving large teams. Games in the genre often have a fairly simple main structure, but with a lot of data entry and scripting involved to create the story, and tools are often created for these jobs. (In addition a lot of the development effort is used in creating visual and audio resources for the games, and in creating the 3d game engines – but a further elaboration on this is outside the scope of this thesis).

Several tools are also available commercially or in the public domain. The earliest example of such tools is probably The Quill[14] – a tool for creating text-based adventure games – which was published in 1983. Game developers tend to want to give their private "touch" to the games though, and as the general tools are often not flexible enough the game developers are faced with either creating their own tools or doing the job without domain specific tools. A recent trend, with the Neverwinter Nights[15] game as the prime example, has been to include the toolsets with the games, allowing game players to create their own scenarios based on the game engine. Although this has been well received in the gaming communities, the toolsets are fairly complex, and much of the behaviour is expressed in scripting languages similar to C, and thus requiring that the scenario builder has at least some programming experience. The tools could clearly be improved by adopting some of the techniques developed in the end-user programming community.

In a typical role-playing game the user controls the movement and actions of one or more player characters, and the role-playing game world is composed of a set of 2d or 3d locations/maps which the player characters move around in. Player characters can meet and interact with non-player characters, and find and trade equipment and other items. Usually the games also have a combat element, where the player character meets hostile non-player characters or monsters that have to be defeated. In multiplayer games the concept is extended with new interaction possibilities which allow the player characters to interact in different ways, e.g., to chat, trade items, collaborate on tasks requiring multiple players, or combat each other.

To keep the example simple enough, focus is here mainly kept on the dialogue part of the game, where the player character interacts (usually talks) with non-player characters in the game. This part of the games often contains most of the story line and scripting, and with the right tools it can be implemented by script-writers without professional programming experience. Also this can be an example of a reusable component which

[12] The Final Fantasy series is developed by Squaresoft. http://www.squaresoft.com
[13] Baldur's Gate was developed by BioWare corp. http://www.bioware.com
[14] The Quill was published by Gilsoft in 1983 for the ZX Spectrum computer
[15] Neverwinter Nights is developed by BioWare corp. http://www.bioware.com

could be of interest across game developers within the role-playing game genre and similar genres (for example, adventure games).

While the style of the dialogues in role-playing games can vary somewhat from game to game, the main structure is usually similar. The player controls the dialogue by selecting from a list of options for what the player character says or does next in the dialogue. As a response to the selection, the non-player character responds, e.g., by providing some information or giving the user some item. The selection also typically results in a new set of options for what the player character can say or do next, and this drives the dialogue on. The options made available in a dialogue are often also depending on what the player characters have done earlier in the game, e.g., tasks performed for the non-player character, but also on the skills and personality which the user has selected for the player character. A simple example from a dialogue in the Neverwinter Nights game is shown in Figure 20; when the user selects the first option in the left screenshot it leads the dialogue on to the state shown in the right screenshot.

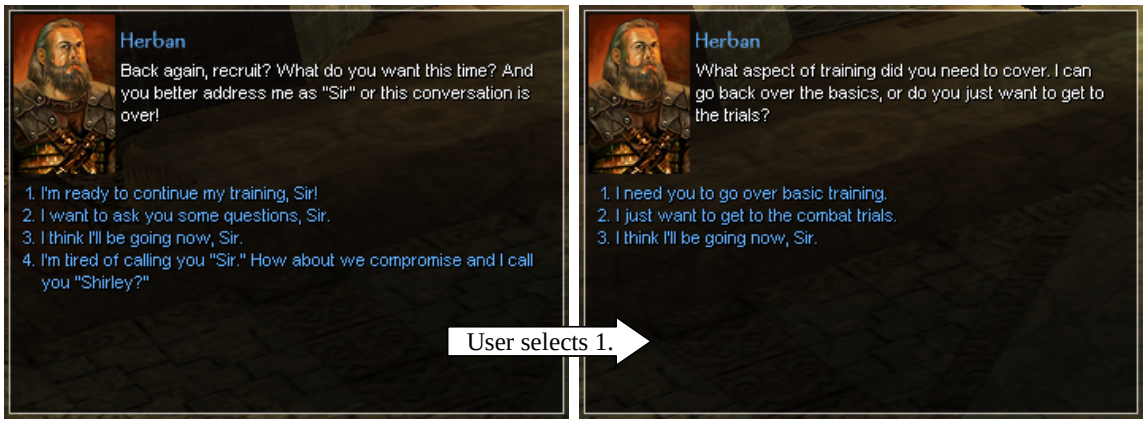

Figure 20 - Example of dialogue from the Neverwinter Nights game

One possible design for describing the dialogue part of a role-playing game is presented in the UML class diagram of Figure 21. In this design, a Person (non-player character) has associated a set of set of interaction cards. An interaction card contains a set of interaction entries, which is the list of possible interactions which the user selects from at a specific point in the dialogue. For further fine-grained adjustments each of these interaction entries can have an associated condition which must be satisfied for that entry to be included in the list of options, e.g., that the player has performed a specific task. An interaction entry also contains the dialogue response of the non-player character, a reference to the next interaction card to use in the dialogue, and it may also contain a sequence of other actions which the selection of this interaction entry triggers. Such sequences may contain textual responses and pictures to be displayed, special actions such as initiating trade between the parties, and scripts in a suitable scripting language, for instance triggering that the user acquires an item, or that the non-player character becomes hostile.

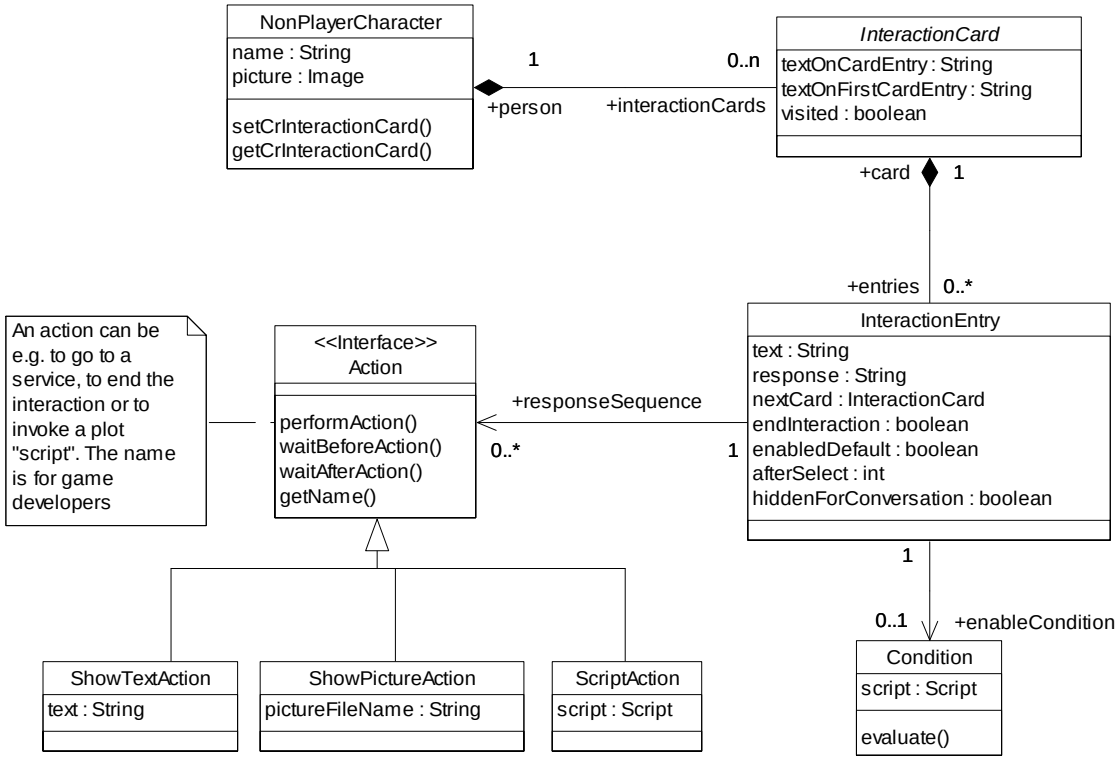

Figure 21 - UML model of the dialogue part of a role-playing game

Role-playing games are good examples of software which, with the right tools, can be developed by users without professional programming experience. An extensible application composition environment for role playing games should thus be well suited for use as an example in this work. Some of the properties of the example are:

- Role-playing games have a fairly simple main structure which can be predefined by a framework and supported by tools. Most of the fundamental and default behaviour is implemented by the framework, and only customised by scenario-specific scripts.

- Applications are built by creating and editing object instances from predefined types, and composing them within predefined structures. Default behaviour is overridden by scripts associated with the objects.

- Document-based persistence is suitable for storing a role-playing game scenario, although a database approach could also be applied. During development of the scenario an open and easily readable file-format, e.g., based on XML could be well-suited, but for distribution of the game an encrypted format is probably wanted to protect the information from unintended use and tampering.

- Components can be used to introduce new object types, both for minor additions such as items supporting a special operation, or for larger additions such as, e.g., introducing trading with non-player characters, or a board-game that can be played against non-player characters as part of the story.

- A predefined runtime system can be made which can be used with any role-playing game scenario, and which can be packaged together with the scenario and any components used by the scenario, and distributed as a stand-alone application.

4.5 Summary

This chapter has presented two case studies based on previous European research projects we participated in. In both cases component-based environments for composing personalised software tools for disabled and elderly users were developed. The experience from the cases is used as input to the development of the methods and tools presented later in this thesis.

The ComLink software allows *integrators* to compose personalised communication tools with alternative access support, and *facilitator* to perform day-to-day minor adjustments to such tools. A personalised tool is composed by building up a vocabulary for the user, arranging a layout using items from the vocabulary, and configuring how the user can navigate and select items in this structure. The ComLink software consists of an extensible set of subsystems, each of which is built as a component framework supporting extensions within the subsystem. This approach allows the environment to be extended with different kinds of components, e.g., new layout elements, vocabulary modalities, and configuration elements as well as new subsystems.

The TASC software allows a *carer* to select among a set of coarse grained components to compose a tool providing cognitive task support for an end-user. Associated with each component visible to the end-user is a corresponding builder component which enables the carer to predefine part of the content for the user and adjust settings for the component.

The last section of the chapter presented an example of a possible new extensible application composition environment for creating role-playing computer games. The example is focused on the dialogue part, which constitutes most of the story in such games, and is the part of the game where most of the scenario specific behaviour is defined (e.g., using a scripting language). This example will be used throughout the thesis to illustrate the proposed frameworks and tools.

Chapter 5 User and Developer Roles

This chapter defines a value chain based on a set of roles involved in development and use of extensible application composition environments for end users, and thus addresses the research question: *Which user roles and developer roles are involved?* Organising the software production in a value chain is a step in industrialising and streamlining the development effort, giving a clear division of labour among the involved roles. The set of roles is identified based on analysis of the case studies, studies of end-user programming tools and considerations on how component software contributes to this picture. Further focus is given to the developer roles most central for this thesis, and the main tasks performed by these roles are identified and described. The tasks with the greatest potentials for reducing the required development effort is in later chapters used as a starting point for proposing frameworks and tools to aid the development effort.

5.1 Overview and Description of the Roles

This subchapter proposes a value chain for the development and use of extensible application composition environments for end users. Figure 22 shows an overview of the value chain formed by the various user and developer roles, and the flow of software in the value chain. A role, in this context, denotes a function performed by a person within the overall development and use of a software system. The roles towards the bottom of the diagram require gradually higher software development expertise. The type of knowledge required is increasingly more specific and concrete for the roles towards the top of the diagram, and this applies for both domain knowledge and software development expertise.

Each of the roles in the value chain is described in more detail below. The roles presented in previous chapters on component software, end-user programming and the case studies were used as a starting point when finding and defining the roles used here. The description of each of the roles includes a comparison to similar roles from these chapters.

In some cases, the same person may play more than one of these roles. Even in these cases, keeping the roles clear in mind is important, as different activities and tools are associated with each role. For instance, even if it is the same developer that develops a domain framework and multiple applications within the domain, it may make sense to

create an application composition environment for the domain to develop the set of applications more quickly.

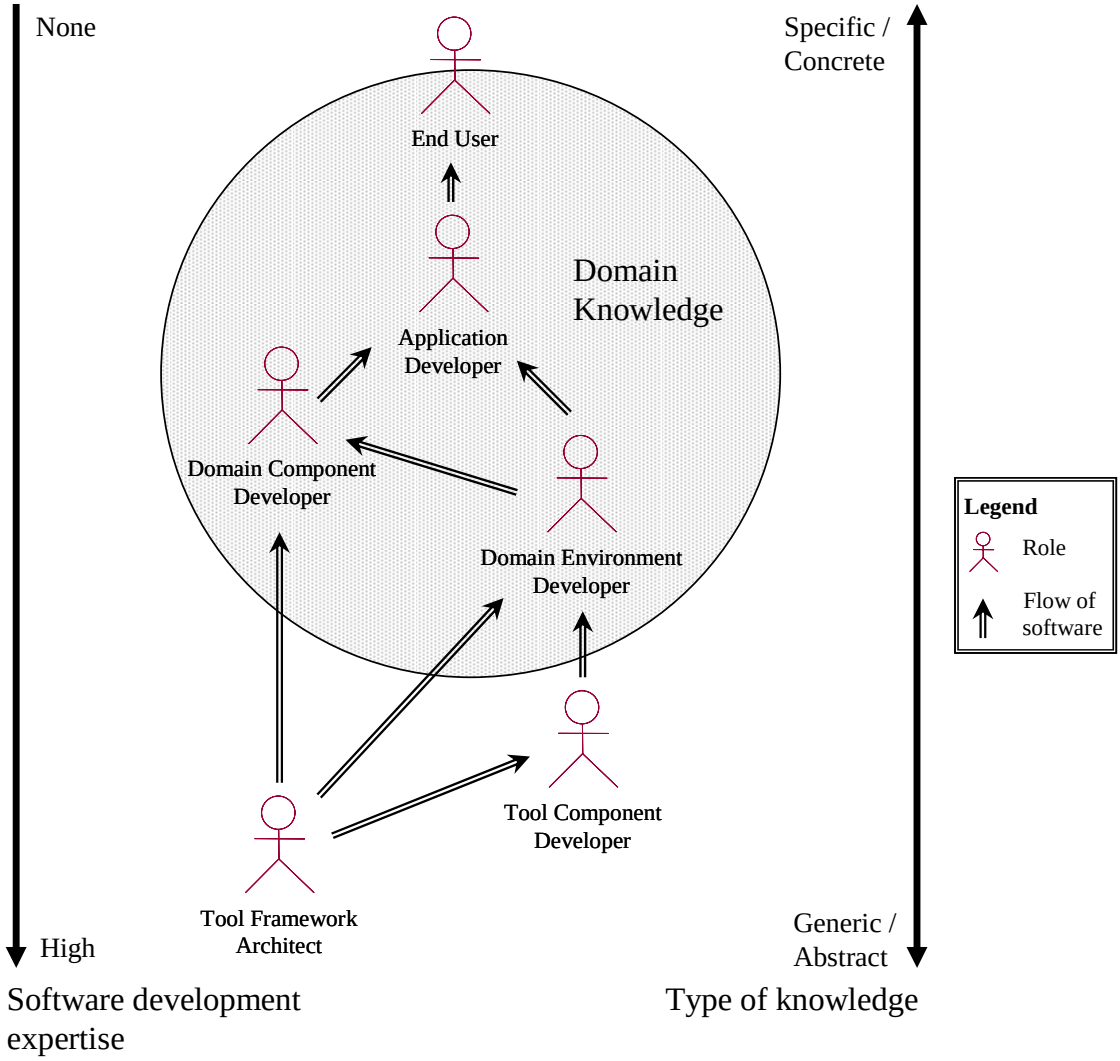

Figure 22 – User and developer roles, and the flow of software in the value chain

5.1.1 End User

This is the user of the application made using the application composition environment. From the domains of the case studies, this may be a physically or mentally disabled user who needs a personalised tool to make life easier, or from the role-playing game example, this would be the player of the computer game.

5.1.2 Application Developer

The application developer is a person with interest and knowledge about a specific domain. This person will not usually be a professional programmer, but may have skills similar to what is described as a “gardener” in [Nardi 1993]. The role matches the

application developer role from end-user programming, and partially the component assembler profession proposed by [Szyperski 1997].

In the domain of personalised tools for disabled users, the application developer is usually someone close to the disabled user who knows the user's personal requirements. This may be a relative, teacher or therapist of the disabled user. Referring to the motivations presented for end-user programming in the introduction of this thesis, the motivation for being involved in end-user programming in this area will mostly be practical and economical.

The application developer in the game domain may range from a game player who wants to start creating games on his/her spare time, to the author of the story line of a game produced by a company in the gaming industry. The motivations here can be philosophical – allowing transition from a passive role of game player, to an active role of creating stories. It also has a practical side of allowing the author to personally develop the story part of the game without the intermediate transfer of the story to a professional software developer.

These two examples of this role demonstrate a potentially large difference in the distance and relation between the developer and user. In the disabled user example, the application is made personally for the user, while in the game example the resulting game may be used by millions of users unknown to the developer.

In the Comspec project, the role of application developer was further divided into the integrator and facilitator roles, the first of which can create a new application, while the last only performs minor day-to-day adjustments. Similar to this, Hypercard for instance has different user levels which define what the user is allowed to do – ranging from simple browsing to full development including editing of scripts. [Stiemerling 1998] describes different tailoring rights. In this work we will not further subdivide the role of application developer, but instead note that it may be a need to support a range of different skills for this role.

5.1.3 Domain Environment Developer

The development of an application composition environment requires professional programming expertise. It also requires a good deal of knowledge about the domain the environment is made for.

In the case of environments for creating personalised tools for disabled users, the environment developer is probably a software company in the rehabilitation industry. In the game domain this may be the tool and framework department of computer game producer, who either keeps these tools in-house, or distributes them along with a game or as a separate product.

The domain environment developer role is similar to the platform developer in Comspec and TASC, and to the environment developer in end-user programming. The main difference is that this role can build on the software provided by the tool framework architect and the tool component developer. The work is thus simplified, and can be focused on developing a domain framework and assembling tool components to a new environment. Compared to the professions suggested by [Szyperski 1997], this

role thus partially overlaps with the component framework architect when developing the component framework for the domain. However, it also performs work similar to the component assembler when assembling new environments.

5.1.4 Domain Component Developer

Extensibility means that new components with new functionality can be added to the composition environments. These components may be developed by the developer of the composition environment, but if the systems are open it can also offer the opportunity for extension with components made by third parties. This indicates a role of a domain component developer. The components can in theory be developed independently and be used in multiple domain composition environments, with the potential of a broader market. However, it is more likely that a component developed within a specific domain framework can be integrated smoothly by the application developer. An alternative to domain component developers directly developing for the domain framework is that someone who knows the domain framework well adapts the components. This work may, e.g., be done by the same person that has the domain environment developer role.

This role is similar to the developer role in defined in Comspec and TASC, the plug-in developer in end-user programming and the component programmer profession proposed by [Szyperski 1997].

5.1.5 Tool Framework Architect

This developer role creates the architecture, frameworks and meta-level tools used by environment and component developers. The work of this developer allows the domain environment developer to create an application composition environment more quickly. The work in this thesis contributes in this area, and focuses on how to make the work for the environment and component developers easier and quicker.

This role takes over part of what was included in the platform developer role in Comspec and TASC, and the environment developer role in end-user programming. Compared to the professions in [Szyperski 1997], it is both a component system architect and a component framework architect.

5.1.6 Tool Component Developer

Additional tools for the domain environment developer can also be created by third parties. This can, for example, be implementation of new visual formalisms, scripting languages or other end-user programming techniques made to fit into the framework used to develop domain specific composition environments. The developer in this role may, e.g., be a company specialising in support of a specific end-user programming technique.

Compared to Comspec and TASC, this role partly overlaps the platform developer role. It is more focused though, with the goal of developing a component for a special technique instead of a full environment. The same also holds when comparing it to the

environment developer role in end-user programming. In term of the professions in [Szyperski 1997], it is similar to the component developer.

5.1.7 Summary and Discussion of the Roles

As stated in the introduction chapter, the main goal of this thesis is to answer the research problem of how we can streamline and reduce the development effort required to build extensible application composition environments for end users. The identification of the roles involved was a first step to answer this problem, as it identifies whom we may need to support the work of and how the division of labour can be improved. A summary of the roles with the software they create and who uses that software is presented in Figure 23.

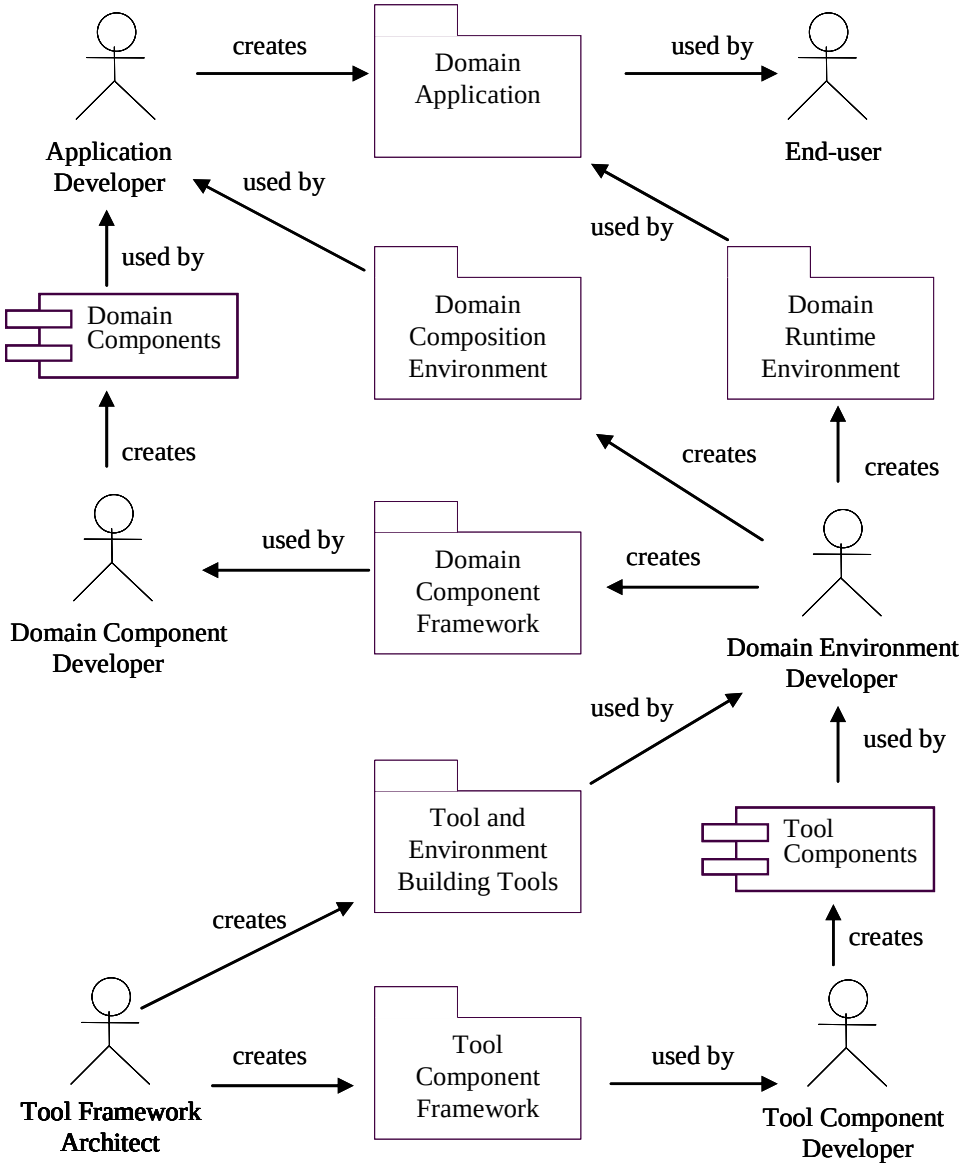

Figure 23 – Creators and primary users of software in the value chain

As is clear from both from this diagram and the description of the roles, these roles do not match one-to-one with the professions suggested by [Szyperski 1997]. One reason

for this is that the professions were suggested in the context of component software in general, while the roles proposed here are specialised for the context of a value chain for creating domain specific application composition environments.

When compared to the roles traditionally involved in developing end-user programming environments, the suggested value chain divides the responsibilities earlier held by the environment developer into different roles that are more suitable for enabling reuse and simplifying development of new environments.

The roles identified for Agentsheets in [Repenning 1993] do not include tool or domain component developer roles. The remaining roles identified here mostly match the roles of Agentsheets, with the Domain Environment Developer corresponding to the Spatio-temporal Metaphor Designer, and the Application Developer to the Scenario Designer.

5.1.8 Roles Focused on in This Work

The domain environment developer has the central role in the development, and will be the role this work primarily focuses on support for. The next section focuses on the tasks of the application developer and how to support these tasks, as this defines the major part of the software the domain environment developer have to build. The domain component developer role is the only other role directly relating to the application developer, and thus the secondary focus of this work is on how to support this role. By selecting these roles to focus on, this thesis can be seen to have the viewpoint of the tool framework architect, and to some degree also of the tool component developer.

Most of the requirements an end user has for a specific application are domain and task specific requirements, and thus these requirements are mostly gathered by the application developer and to some extent by the domain environment developer. Of more general requirements, the end user usually will need a way to make the status of the application persistent. This thesis work does not focus further on the end user role.

5.2 Tasks of the Application Developer

The focus here is on the domain-independent tasks of the application developer role, and the software that covers these. The tasks are presented as use-case diagrams with associated textual description. In addition to the tasks presented here, an application developer will have domain-specific requirements. Based on these the application developer should select a suitable environment for the application domain.

The general tasks described here have been extracted from how applications are created in the end-user programming systems presented earlier and in the case studies. The main use-cases that the domain composition environments should support the application developer in are presented in the fist subsection and Figure 24. The use-cases ***edit application*** and ***manage components*** are further detailed in sub-sections and the use-case diagrams in Figure 25 and Figure 26.

5.2.1 Main Use-Cases

Most end-user programming environments are document-based, and in these to ***manage application versions*** usually just come down to facilities for saving and loading the application edited within the environment. If there is more than one developer involved in editing the application it may be useful to have more support in this area. Also, if more complex persistence schemes are required, this use-case may need to be detailed. In this work we will settle for the simple document-based storage usual within most end-user programming environments, and further detailing is not necessary.

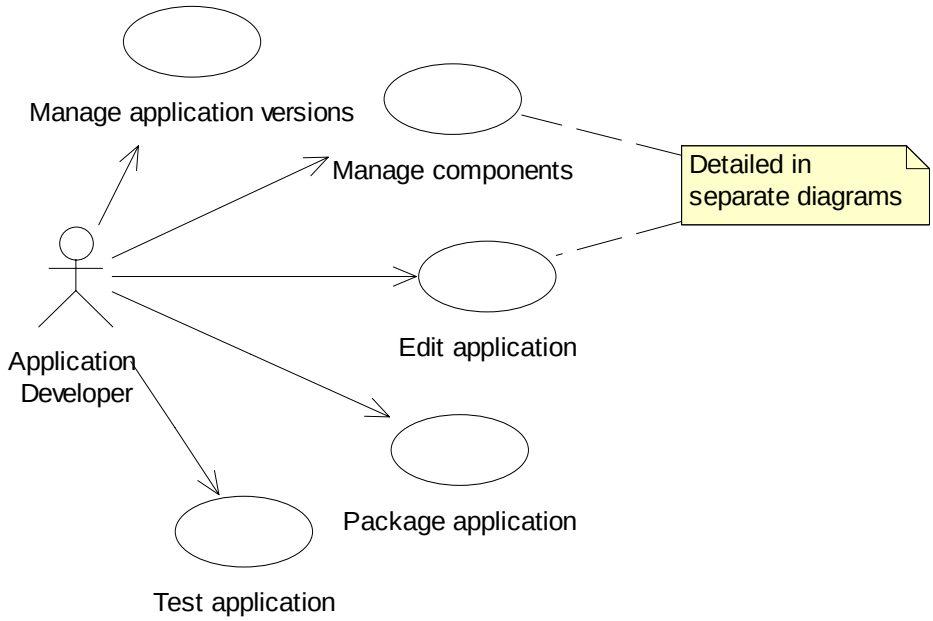

Figure 24 - Overview use-cases for the application developer

The application developer needs to be able to try out and test parts of or the whole the application. This is covered by the ***test application*** use-case. In addition to the runtime environment seen by the end user, when trying out the application it may be useful to have debugging tools that, for instance, allow inspection and editing of application state during the execution. Various testing tools could in addition help finding consistency problems of domain specific or more general nature.

The ***package application*** use-case marks the activity of wrapping up everything that is needed to run an application. This includes the runtime part of the domain software, any components used in the application, the application files and any local resources (pictures, sounds, etc.) that are referred to from within the application.

5.2.2 Detailing of the Edit Application Use-Case

Figure 25 describes the use-case of editing an application in more detail. In traditional object-oriented software development by professional programmers much of the focus is on developing the collaboration among the classes, organising class hierarchies, and implementing the classes. The focus of end-user programming environments is often more directed at the instances: creating object instances and setting their properties, and

defining the special cases of behaviour. The added behaviour may affect only a single object, or in some cases a group of objects.

The use-case ***create instances within structure*** covers how the application developer creates new objects. The objects are created within some containing structure. For instance in the grid-based environments (and spreadsheets) an object is created within a cell, while in the Comspec layout an object is created within a layout container etc. In a simple case there may be only a single structure editor, e.g., a grid, which organises the whole application. In more complex settings a set of different structure editors will be needed (for instance, Comspec have a layout, configuration, vocabulary and toolbar). Each of these structure editors should only allow creating objects that are meaningful in the context.

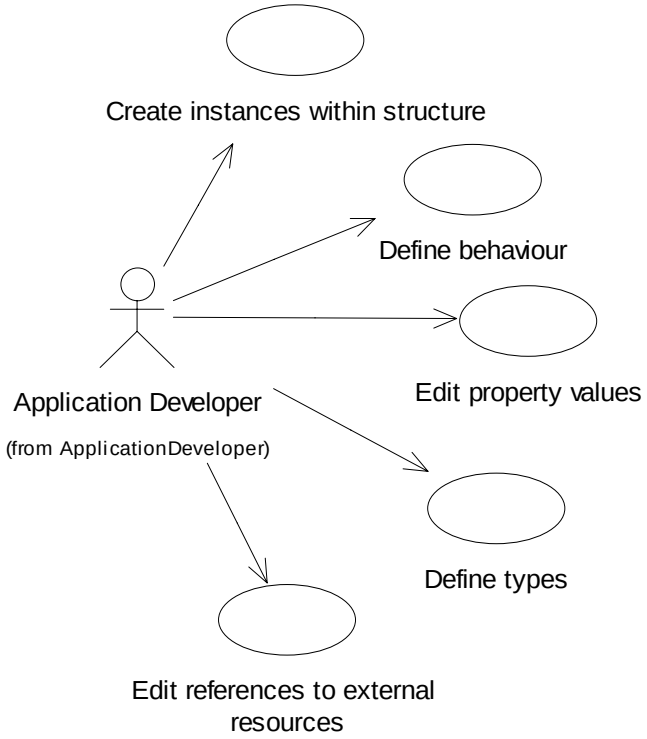

Figure 25 – Detailing of the edit application use-case for the application developer

The ***define behaviour*** use-case is affected by which approach to definition of behaviour that is supported. In scripting approaches the behaviour is often associated with some kind of events for an object (or its class) which triggers the script. Rule-based systems can associate the rules with an object that is involved (like it is done in the grid-based environments), or the rules can be of a more global nature.

It may be necessary for the application developer to define new types. This is covered by the use-case ***define types***. This may range from defining completely new types to just adding a new property. As these types do not have any special handling in the predefined runtime system, the use and influence of these types are determined by the behaviour of any base type they are derived from, and from the special cases of behaviour defined by the application developer.

The use-case ***edit property values*** covers the activity of filling in the content of the domain objects. A property value can be a simple value like a number, a text or boolean,

or a simple object like a colour that is contained by the domain object. However, it can also be a reference to another domain object. To support editing of properties, a common approach in development environments is to provide a property sheet. In such sheets all properties of an object are listed by name, and a simple editor is provided for each property. Another approach is to have a more specialised detail editor for each domain object type created to better support the editing operations of the application developer. This was the approach followed for the objects editable by the facilitator role in Comspec. In addition editing of some property values may also be integrated more closely in some of the structure editors. A layout editor will, e.g., edit the size and position properties, and may also allow alternate ways of modifying other display properties like colour, font etc.

The last use-case in the edit application diagram is ***edit references to external resources***. This is similar to ***edit property values***, except that the property is set to refer to an external resource, e.g., a picture or sound file. Editing of these will usually be done together with other properties, and often involve selection from a file dialog. The reason this is covered by a separate use-case is that the semantics of referring to an external resource is different than editing a self-contained value. When the application is to be deployed on another computer, these external resources will have to be copied too. In addition to the editing support, it may be of interest for the application developer to get a viewable list of all referred resources within an application for overview.

5.2.3 Detailing of the Manage Components Use-Case

A domain specific application composition environment is supposed to cover most of the needs of the application developer. In some cases though, the application developer may want or need some extensions to what is originally included in the environment. This may be components that cover functionality needed only in a few cases, or extensions which the environment developer did not envision but where a component developer has covered the gap. Figure 26 is an overview of the activities an application developer may be involved in when managing components for an application in an environment.

The ***find component*** use-case may not necessarily be supported by the environment, but the functionality would be helpful for the application developer. With the widespread access to internet, support for these kinds of activities has recently become more usual in professional development environments; for instance, Sun’s Forte for Java can check to see if there are any new components for the environment, or updates to existing components (see Figure 34 in the next chapter). Application development environments may support this activity by providing lists of available components that is suitable in the environment, including descriptions of what each component will do and easy access to buy or download free components.

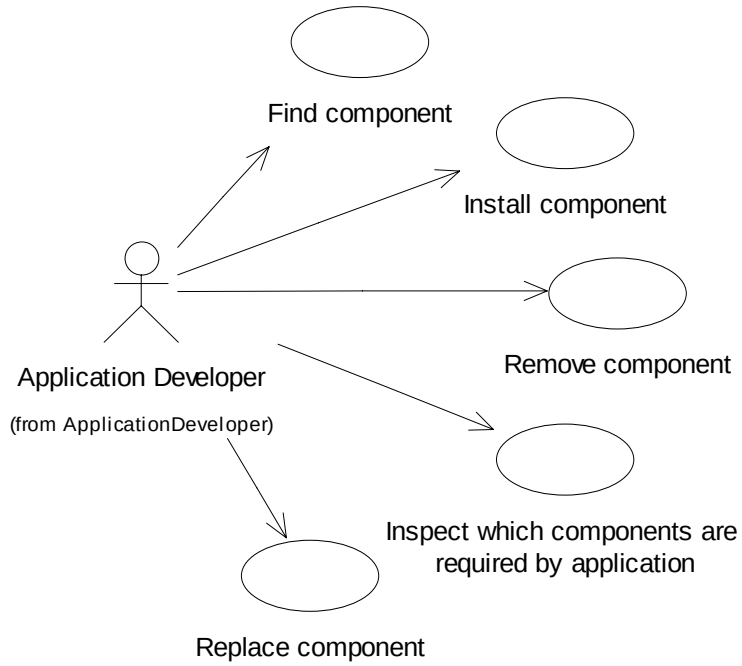

Figure 26 – Detailing of the manage components use-case for the application developer

To enable extension it is at least necessary to support the ***install component*** use-case. The realisation of this can be as simple as having the developer put a deployable component in a special folder, or it can be integrated with the support for ***find component*** that was described above, and allow installation right after downloading. Installing a new component frequently will add new domain object types to the system, and after installation these should now be available for the ***create instances within structure*** use-case (described in 5.2.2). The environment may support extension either at the environment level, where the extension is made available to all applications made in the environment, and/or it may support extensions specific to an application.

Preferably the ***remove component*** use-case should also be supported by the environment. There are two issues involved here: removing the component from the environment so that it no longer allows creation of new instances of domain objects supported by the component, and removing the uses of the component from an application. This last issue is more demanding to support, and also the effects on the application may not be obvious. One approach would be to allow removal of the component only when it is not in use, and require manual removal of uses of the component.

The ability to ***inspect which components are required by application*** is useful for a developer who takes over the development or maintenance of an application made by someone else, and can also be useful when development for other reasons are moved from one computer to another.

A special case of installing a component is the use-case ***replace component***. This may be supported by the environment, e.g., to allow application developers to replace an existing component with another component that has an implementation that for some reason is better for the application developer than the original component.

5.3 Tasks of the Domain Environment Developer

The domain environment developer creates software that is used by two of the other roles: the application developer and the domain component developer. The application developer should receive software that supports the use-cases he/she is involved in, including a runtime system that will be used in the end user application. The domain component developer needs to receive some information and software to be able to create components that can extend the domain environment.

5.3.1 Overview

Figure 27 shows an overview of the main activities of the domain environment developer. While the use-cases presented for the application developer focused on the activities performed within the application composition environment, the use-cases presented here covers the broader set of development activities required to develop a domain composition environment.

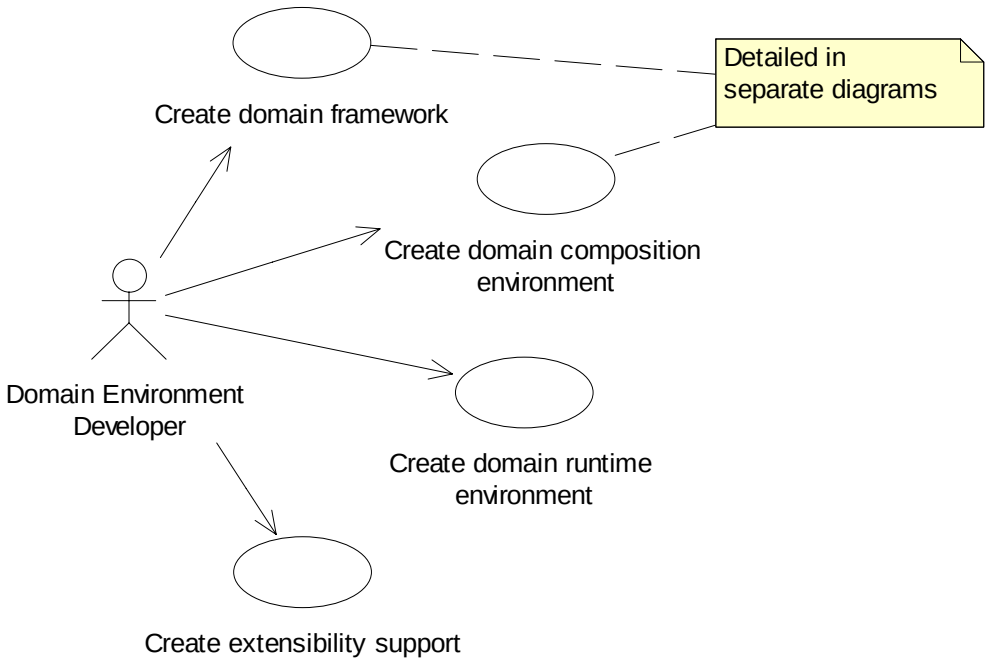

Figure 27 – Overview use-cases for the domain environment developer

Chapter 3 introduced the concept of a component framework as an approach to build extensible software systems. The component framework defines the interfaces and rules of behaviour that allows the components to collaborate. Thus a domain specific component framework is a suitable approach to create extensible domain composition environments. The ***create domain framework*** use-case included in Figure 27 is detailed further in Figure 28.

The domain framework will also be a prerequisite for the work in the other main use-cases. ***Create domain composition environment*** is the activity of creating an environment that is suitable for the application developer based on the domain framework. This is detailed further in the use-case diagram of Figure 29.

Create domain runtime environment represents the work of creating the environment that will ensure the execution of the completed application for the end user. This includes configuring execution environments for the behaviour specifications that the application developer will use, setting up the main flow of control and the predefined parts of the user interface of the application etc. This work is not detailed further here, as the main focus is on the composition environments and their extensibility.

While the foundation for the extensibility of the environment is provided by the domain specific component framework, the use-case ***create extensibility support*** covers the activity of making sure that the domain component developer receives what is needed to easily create components for the environment.

5.3.2 Detailing of the Create Domain Framework Use-Case

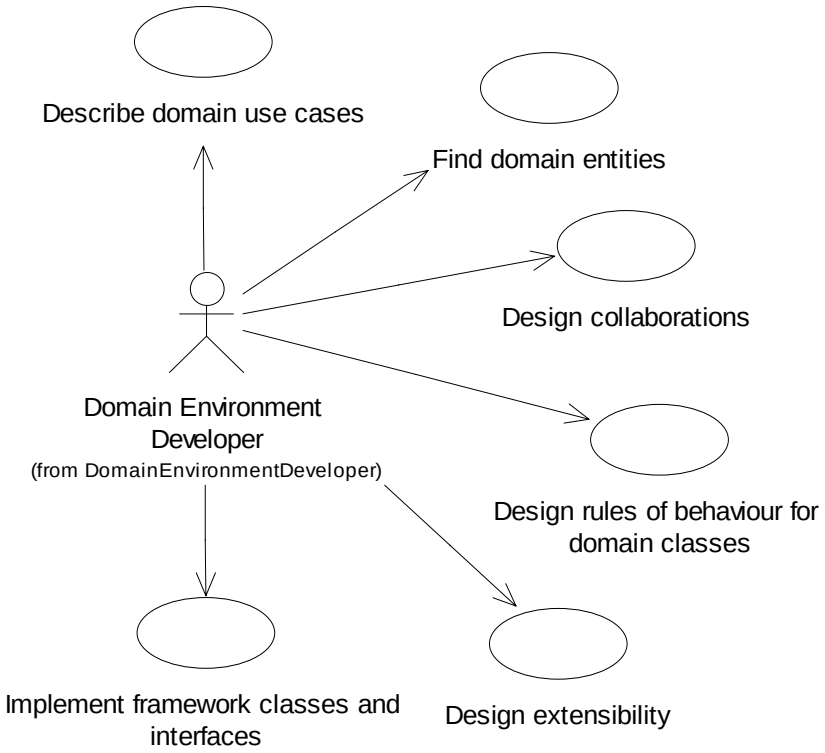

Figure 28 - Detailing of the create domain framework use-case for the domain environment developer

Figure 28 presents a use-case diagram for creating a domain framework. Analysis and design methodologies like OORAM [Reenskaug 1996] and Rational Unified Process are examples of approaches that are suitable for the difficult work of creating a framework, but creating good frameworks still partially remains an art. The use-cases in the diagram are activities that are usually part of the work of creating a framework, although how the work is done varies with the methodologies. Some of the design activities can be supported by modelling tools like Rational Rose, while the implementation partly can be supported by code generation tools. Also the component system developed by the tool framework architect gives a foundation for the

development of the framework, and guides the developer in the decisions on architecture, layer model and how to support extensibility.

5.3.3 Detailing of the Create Domain Composition Environment Use-Case

A major part of the work of the domain environment developer is covered in the use-case ***create domain composition environment***, which is presented in Figure 29. This work includes making sure the use-cases of the application developer is supported by the environment. Creating this support from scratch is time-consuming, and it is a goal that the frameworks, tools and components created by the tool framework architect and the tool component developer simplify this work to become a configuration task.

The use-cases of Figure 29 include tasks for creating support for the use-cases of the edit application diagram (Figure 25) for the application developer. The ***configure structure editor*** use-case is part of the support for the ***create instances within structure*** use-case from Figure 25. Suitable structure editors have to be selected and configured. The configuration of a structure editor includes selecting which classes can be created within it, and how. The specification of which classes to include can be in the form of all concrete subclasses of a specific class or all classes supporting a set of interfaces etc.

Configure property editors for domain classes supports the use-cases ***edit property values*** and ***edit reference to external resource*** from Figure 25. As already mentioned property-sheets and custom made detail editors (customisers) are two existing methods for editing properties, of which custom made detail editors are preferable for the end user. An approach to reduce the effort involved in creating custom editors would be to support configuration of such editors instead of requiring a programming effort. This would reduce the flexibility somewhat, but could be combined with allowing programmed custom detail editors to be used where present. For the references to external resources the domain environment developer needs a way to mark these, so they can be automatically included in the management of such resources.

The ***configure points of customisable behaviour*** use-case includes support for the use-cases ***define behaviour*** and ***define types*** from Figure 25. There are many different techniques and representations used for defining behaviour, as was described in the end-user programming chapter. These include scripting, flow-charts, data flow, etc. In some cases the definition of behaviour will most naturally be done within the same editor that defined the objects. Behaviour editors may for instance organise existing objects within the system like a connection editor used to connect senders and receivers of events. In this case the developer has to specify which objects in the context that can be used. The environment developer must also decide to which degree the application developer will be allowed to define new types and/or use components that extend the available types, and where in the system these types can be added.

The support needed by the application developer for the remaining tasks of creating a domain composition environment, such as managing components, managing application versions, testing applications and packaging applications will be quite similar be between different environments, and are thus good candidates for reusable support by configurable standard components of the environment. Such components are created by the tool framework architect or tool component developer, and the domain environment developer need only configure these to suit the needs of the environment under

construction. These activities are covered by the ***configure standard environment components*** use-case.

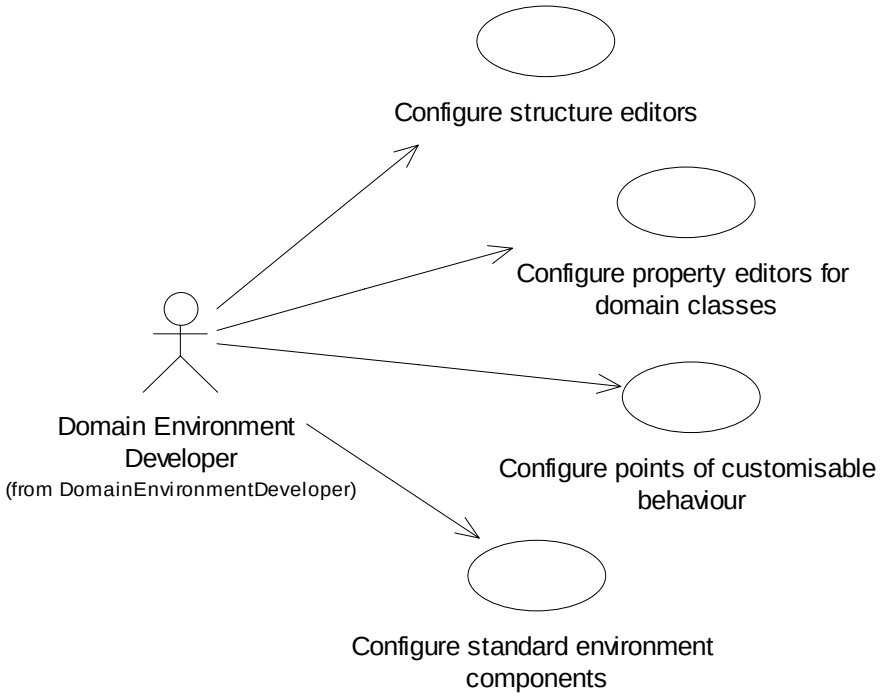

Figure 29 – Detailing of the create domain composition environment use-case of the domain environment developer

5.4 Tasks of the Domain Component Developer

The domain component developer extends the domain application composition environment, usually by extending the range of domain object types. The main tasks of this role are presented in the use-case diagram of Figure 30. For the ***design component*** use-case the developer is guided by the domain component framework, and a modelling tool such as Rational Rose may be used to support this work.

The domain component developer must ***implement runtime support*** for how the component should behave in the domain application, governed by the rules of the domain component framework. Also in the ***implement editing support*** the domain component developer must create an editor for the component, and this work is similar to the ***configure property editor for domain classes*** use-case of the domain environment developer. The ***package component*** task should be quite similar for all component developers, and thus tools for this work could be provided by the tool framework architect.

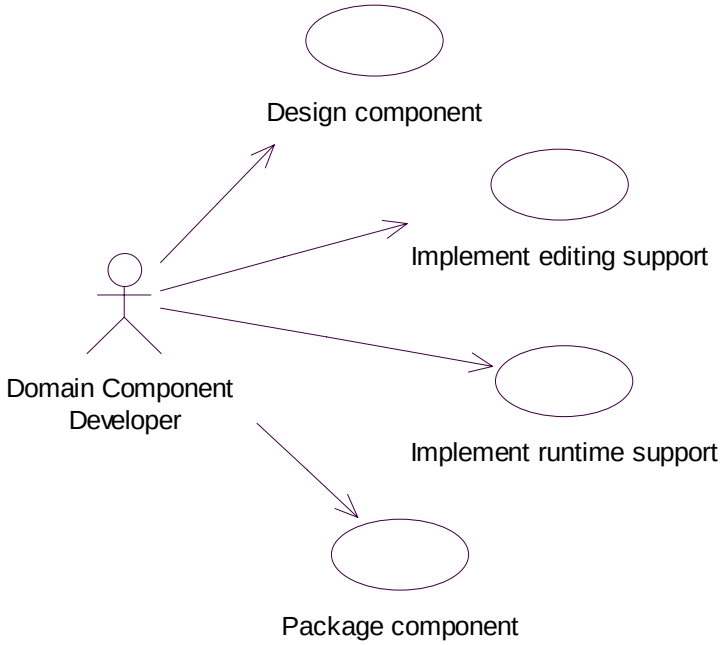

Figure 30 - Overview use-cases for the domain component developer

5.5 Tasks of the Tool Component Developer

Tool component developers develop a range of visual formalism and end-user programming techniques that can be used by the domain environment developer. On an overview level the tasks of the tool component developer is similar to those of the domain component developer in Figure 30. The difference is mainly in the kind of components developed. While the domain component developer mainly focuses on creating a component that fits in the domain framework, with some additional work for fitting in the editing environment, the tool component developer creates domain-independent components with greater challenges on how to fit into the editing framework and the execution model. Examples of some of the component types developers in this role may develop are given in Chapter 6 where the reuse potential of existing environments is examined.

5.6 Tasks of the Tool Framework Architect

The goal of the tool framework architect is to provide a foundation that supports the work of the domain environment developer, the tool component developer and the domain component developer as much as possible. The use-case diagram of Figure 31 gives an overview of the tasks involved.

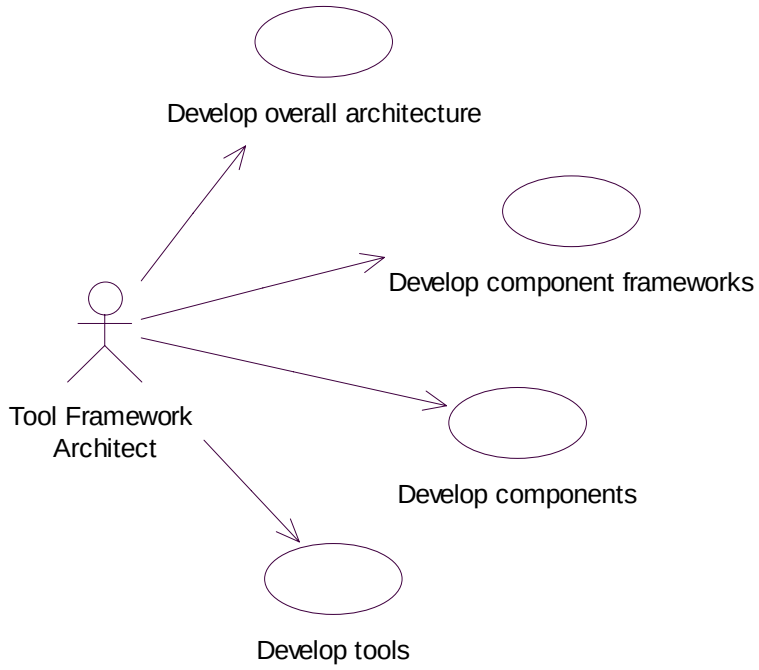

Figure 31 - Overview use-cases for the tool framework architect

The ***develop overall architecture*** use-case covers the work of developing an architecture which identifies how the different component frameworks, tools and component kinds relates to and builds on each other to form a domain composition environment. ***Develop component frameworks*** deals with the work of developing component framework for the editing of applications and for the runtime execution of the finished applications. This work is described in more detail in Chapter 7.

In addition to creating the framework, the tool framework architect must also ***develop components*** that deal with the most common editing tasks and runtime support. Also, the architect must ***develop tools*** that support the tasks of the various developer roles. This work is primarily described in Chapter 8.

5.7 Summary

In this chapter, a set of user and developer roles are identified, and it is shown how these can be connected in a value chain for extensible application composition environments for end users. The roles identified were tool framework architect, tool component developer, domain component developer, domain environment developer, application developer and end user. Of these, the main focus of this thesis is on how to support the domain environment developer, and secondarily; the domain component developer. The tasks of each role (except the end user) were then briefly presented using use-case diagrams along with textual descriptions.

The roles and use-cases presented in this chapter are frequently referred to in the following chapters.

Chapter 6 Assessment of Reuse Properties of Existing Environments

Reuse is a well-known approach to reducing the effort involved in developing software and in increasing quality, and was introduced in Chapter 3 with a focus on software components. As presented in Chapter 2, there is a wide spectrum of available end-user programming environments. These environments are sources of tools and techniques for end-user programming that have proven to work well. Similarly the environments developed in the case studies described in Chapter 4 are sources for reuse. This chapter analyses some of the tools and techniques from these environments that are good candidates for creating components that can be reusable in new environments, and thus answers the research question: *What kind of reusable parts can be identified from existing end-user programming environments?*

The assessment given in this chapter is limited to the environments already introduced in previous chapters. The analysis is primarily done from an external viewpoint, and in most cases without detailed knowledge of how the environments are actually implemented. The assessment is divided into different areas of concern the environments deal with, each of which is described in a separate subchapter. Even though the concerns are presented separately here, there may be dependencies between concerns in the solutions chosen in a particular end-user programming approach, and one component may well cover more than one of these areas of concern. After each area of concern has been presented, a subchapter also briefly discusses reuse and integration across components.

At the end of each subchapter presenting an area of concern, a table is used to present a summary of the reusable parts that were discussed. These tables can be considered as a proposed draft for a reusable part catalogue for building extensible application composition environments. In the tables the type of reuse most likely or most suitable for each part is indicated, differentiating between three broad types of reuse: reuse of concepts and ideas (e.g., patterns and other reuse without code), reuse of source code, and reuse of binary components. Identified dependencies on the other areas of concern are also specified in the table, and in some cases additional notes provide examples or other clarification.

The following list introduces each area of concern and how it relates to the use-cases associated with the application developer from Section 5.2:

- *Language and translation*. This area of concern deals with part of the foundation necessary for supporting the *define behaviour* and *define types* use-cases of the application developer.

- *Runtime behaviour representation and execution*. Includes the support necessary for executing the behaviour defined by the application developer at runtime and part of the support needed for trying the application in the *test application* use-case.

- *Behaviour editors*. Deals with how behaviour is created, viewed and edited, and thus provide the application developer's interface to the *define behaviour* use-case and partly to the *define types* use-case.

- *Structure editors*. Allows the application developer to create instances, and in some cases new types, and to organise these within some structure. Supports the *create instances within structure* use-case and part of *define types*.

- *Detail editors*. How do the environments deal with editing the properties of instances, and thus how do they support the *edit property values* use-case? In some cases detail editors also allow adding new properties to a type, thereby supporting part of the *define types* use-case.

- *Debugging and testing tools*. Tools dealing with finding and correcting problems with the application. Supports the *test application* use-case.

- *Component management*. Some of the environments allow the application developer to add extensions that extend the set of object types available or add new functions. Included here is support for the *manage components* use-case.

- *Packaging and deployment*. Deals with how to support creating deployable applications, supporting the *package application* use-case.

6.1 Language and Translation

The end-user programming environments and the environments from the case studies in the previous chapters display a variety of ways to specify behaviour. The simplest approaches allow behaviour to be expressed just by making connections between objects. These connections can be:

- expressed as direct reference to another object and accessed as a property of the source object in the connection;

- expressed directly in the event model of the component technology in use;

- expressed in a custom made port model like the configuration in the ComLink software.

For more complex behaviour a programming language is usually used to express the behaviour. In some environments multiple languages are used. For example, Microsoft Excel includes a formula language for handling the calculations and uses Visual Basic for Applications (VBA) for more general event handling, macros and scripting. The programming languages used in the environments vary widely in style and expressive power, and the range includes imperative, functional, concurrent, rule-based, data-flow-based and object-oriented languages.

A common feature of most of the behaviour representations, though, is that they have some concept of object. The only exceptions to this among the environments presented in the previous chapters are the original Logo, Lego Mindstorms and the spreadsheet formula language. The support for objects varies in a range from access only to predefined objects to allowing definitions of new classes including inheritance. The reasons for the widespread use of objects is probably both that it is a natural way for handling the visual items that are used in the systems, and that integration with other systems currently is most easily done using the standard object models (COM, Java and CORBA). This will also help with integration during reuse of the language in new environments.

Associated with each programming language usually is a set of tools used to check the source code for errors, and/or to translate it into the appropriate runtime representation. In a translation process the code is made more appropriate for execution, and parts of the source code that is only there for the developer's convenience may be removed – e.g., text comments or layout information in visual languages.

A programming language with its associated tools may seem like an obvious source for reuse in new environments. On second thought, though, the reuse potential here, at least for binary component reuse, is limited to the most generic languages (like VBA). Languages that are either domain specific or tied to a specific visual formalism are only directly reusable in the same domain or within the original visual formalism. As domain specific languages seems to be preferable [Nardi 1993] there seems to be a conflict between the usability of the language and its reuse potential across domains. Fortunately programming languages is a well defined field in computer science, and a lot of good tools exists that aid the development of new languages. There is also an active research community on domain oriented programming languages. Further details on language definitions and translation tools are considered to be outside the scope of this thesis.

Table 3 - Summary of language and translation reuse

Part (broad categories)	Type of reuse	Dependencies	Notes
Language definition	Conceptual	-	-
Translators from source to runtime code	Component or source code	Language definition and behaviour representation	Easier to reuse for generic language
Tools to check source code, e.g., syntax checker	Component or source code	Language definition	Easier to reuse for generic language

6.2 Runtime Behaviour Representation and Execution

The actual internal representation of the code that is used at runtime may vary among the different approaches. The following examples demonstrate parts of the range of possibilities. Behaviour can be represented as:

- a property of a source object in a connection;
- an object structure similar to the UML model for grid-based systems presented in the end-user programming chapter;
- text for interpretation;
- intermediate code (p-code, byte-code etc) for an interpreter or virtual machine;
- native code.

For all but the simplest representations of behaviour a runtime system is needed. Depending on the representation, the runtime system may include an interpreter or a virtual machine. These are components which can be reusable in other environments. One of the problems that must be solved when reusing the runtime systems is how to integrate its execution model with the environment it is reused in. For instance, the runtime environments of the grid-based systems are based on the set of rules being checked at regular time steps, and may require a thread used especially for this purpose. In constraint-based systems this may be even more complex, however none of the environments presented in the previous chapters were based on constraints.

Table 4 - Summary of runtime behaviour and execution reuse potential.

Part (broad categories)	Type of reuse	Dependencies	Notes
Interpreter or VM for intermediate code	Component	-	-
Interpreter or VM for source code	Component or source code	Language definition	Easier to reuse for generic language
Other runtime environments	Component or source code	-	-

6.3 Behaviour Editors

As part of developing an application the developer needs support for:

- creating new behaviour;
- viewing the current behaviour;

- editing behaviour that has previously been created.

In most environments the support for creating, viewing and editing of the code is closely related, but in some approaches the issues are more separated. The behaviour created by a behaviour editor is associated either with an instance or a type, depending on the environment and the behaviour representation used.

For scripting languages and other textual languages the code can be created, viewed and edited directly in a text editor. Text-based code editors are increasingly enhanced with facilities that assist various aspects of the coding. Support for creating and viewing code may, e.g., include:

- templates for event handlers etc.;
- formatting and highlighting of structure;
- code completion schemes (for example, selection from available messages).

Simple text editors are present in almost all software development toolkits. Beyond the text editing features these provide, a reusable text-based code editor should include customisable coding assistance (with features such as examples in the list) to support integration in different environments and languages.

Macros are usually created by recording scripts from a sequence of normal user events. This creates a gap between the creation of the macro, and subsequent viewing and editing of the macro in its script form. The recording capability can be viewed as a front-end to the scripting system that shields the user from the scripting system during creation of the macro. Editing has to be done on the script level, or by completely replacing the macro by recording a new sequence of user events. A macro recorder may be interesting as a reusable part, but it requires that the environment it is reused in produces recordable user events.

In programming by example (PBE) approaches there is a similar situation as for macros. The basic problem is that it is hard to visualise the sequence(s) of user events that the example consists of. In some PBE approaches the existing code is not visualised at all, except for in the behaviour of the program when it is executed. In other approaches it is visualised only as a script.

Stagecast Creator's approach to visualisation is among the best so far in PBE, as the rules are displayed in terms of the visual formalism. For simple examples the 'before' and 'after' pictures of a set of cells with characters show all that has changed. For more complex examples this 'summary' can be opened to show each step that was performed. This list of steps is a bit more abstract, but still closely relates to the visual formalism. From this environment (and similarly from Agentsheets) the rule editor can potentially be reused in new environments. Part of what makes these rule editors appealing is the strong tie to their grid organisation, but the general parts that allow editing of a list of preconditions and a list of actions to trigger could also be used in another context. And so could also a 'summary' similar to the before and after pictures which highlights the important state and state change in the context. In addition to the rule editor, when creating a new rule Stagecast's PBE system also allows manipulation on objects directly in the grid. This can be viewed as PBE "front-end", although the process is quite similar

to creating the rule in the rule editor. This “front-end” is probably only reusable with a grid visual formalism.

In the PBE environment Gamut there is no visualisation of the behaviour. The user creates the behaviour by giving one or more examples. The behaviour can be edited by giving more examples, or by first using the “stop that” button and giving an example that corrects the behaviour. Gamut is domain-specific by focusing on games, and off-the-shelf reuse probably requires use of its visual environment. Gamut (and other advanced PBE systems) uses AI techniques for extracting and combining behaviour from multiple examples. Parts of the AI implementation could probably be reused in a broader context through white-box reuse.

The flowchart editor used for creating behaviour in the Lego Mindstorms software could be reused in other contexts. The domain dependence here is in the commands that are available, and in the conditions used, e.g., to trigger each sequence of commands.

The connection-based editors used in Sanscript, Prograph and the ComLink configuration are general enough to be reusable in many contexts. There are some differences between these editors, some mostly syntactic while others semantic. In Sanscript and ComLink the flow is generally from the left to the right, while in Prograph from the top and down. In Sanscript and Prograph it is possible to decompose a problem by allowing an element in one diagram to have its own internal dataflow diagram that makes up its behaviour. In ComLink it is not possible to decompose a configuration, but in a ComLink application it would probably not be required either, as the configuration is only one part of the whole application. Further, in Sanscript and Prograph the compatibility between a sender and a receiver port is based on the data that is flowing, while in ComLink there has to be a match between a sender and receiver protocol that allows a flow of messages. These differences should be considered when creating a reusable connection editor component, and could be supported by preparing the component for adjustments or extensions.

A general issue for behaviour editors is that there is a dependency between these and the source language. Some of the information that the user enters, e.g., visual cues like position of code elements, formatting, comments etc., are there only for the user’s convenience and has no influence on the behaviour that is created. This information must either be supported by the language or the editor will have to create a system for associating this extra information with the source code.

Table 5 - Summary of behaviour editors reuse potential

Part	Type of reuse	Dependencies	Notes
Textual code editor with coding assistance	Component or source code	Language	Easier to reuse for generic language
Macro recorder	Component	-	Require recording of user events
PBE recorder	Component or source code	Collaboration with structure and	-

		detail editors	
Rule editor	Component	May require collaboration with structure and detail editors	-
Data flow editor	Component	-	-
Flowchart editor	Component	-	-
Collaborating object selector	Component	-	Require naming or other unique visual identification of objects

6.4 Structure Editors

Most of the environments described in the previous chapters have one or more editors that are used for creating various parts of the content. In end-user programming predefined structures of data instances often have a more central role than in other forms of programming.

A kind of editor that is common in many environments is a user interface editor. This is usually an object-oriented drawing editor that in addition has support for defining event handling etc. A palette containing the various object types supported by the editor is used to allow the user to create new objects. The editor may allow hierarchical layouts to be built by allowing a set of containers. Some of these containers may also display only part of their content, either through scrolling or through multiple pages/cards like the card stacks in HyperCard and ComLink and the tabbed pages used in many environments. User interface editors are examples of editors that allow the developer to arrange objects in a structure. In some environments the user interface editor has the only structuring mechanisms for the developer. This is, for instance, the case in HyperCard where the cards and backgrounds used for structure and in Gamut where stacks are used similarly.

A user interface editor usually manages a set of relations among the elements of a layout such as (hierarchical) containment, visual order, sequencing (e.g., tab order) and role in UI-event system. Also the editors often include tools affecting some of the properties of selected object, such as aligning positions, changing colours and fonts etc.

There are differences between the policies of UI editors, and the features the editors have. Some of the variations are:

- allow hierarchical content or not;
- supported layout constrains;

- allow overlap of contained content or not.

In professional development environments user interface editors are often tightly integrated with the behaviour editors; a double click on one of the objects in an interface often opens directly the code of the event-handler for the object. For the interface to do anything useful it is usually necessary to handle many low-level events in this way. For an end-user programming environment the actual visual editing of the interface objects can be quite similar, but preferably much of the low level event-handling should be avoided. To make this possible the object types available to compose the interface from need to be "smarter" and probably more coarse-grained than the usual simple widgets.

An editor type that is closely related to the user interface editor is a grid- / table-based editor. This kind of editor is used, for instance, in the grid-based systems AgentSheets and Stagecast. Also the table of a spreadsheet is very similar to this. This kind of editor supports a set of relations:

- neighbour cells;
- same column, same row;
- distance.

The actual grid/table may be visible or invisible at runtime, but in the most prominent examples (mentioned above) it is also the runtime user interface, and thus visible. An important difference between the grid-based environments and the table-based, is that in the grid editor all cells are of equal size. This is of importance at runtime, as grid-based content often are allowed to move around in the grid.

A tree-based editor gives a general way to manage a hierarchical structure, and is used in many existing user interfaces. A well known example of its use is the Explorer in Windows. It is also used in development environments, e.g., for the explorer used for file systems, project structure and runtime objects in Forte for Java (see Figure 32), and in the main view for the model structure in Rational Rose. For the hierarchical structure, the tree-based editors arrange the containment relation. One of the strengths of the tree as view/editor is that it can organise a large structure and still give a suitable overview. The tree editor naturally supports a set of restructuring operations. Through drag and drop elements in the tree (with their sub-content) may be moved to another position in the structure. Tree views are often used in combination with other editors, and are then usually the main structure editor.

Both in the ComLink and TASC environments tree-based editors were used. The ComLink vocabulary editor is used to create a set of vocabulary items (words or expressions from the end user's terminology), and these items are organised in groups, where each item can be assigned to one or more groups. Each item can hold one or more modalities, for example, bliss symbol, text, picture etc. The editor used for creating the task templates in TASC, is quite similar in structure. Here tasks are organised in groups, and although a task could not belong to multiple groups in the software that was created in the project, this could be a natural extension. Each task can again consist of other tasks, recursively. In both cases the complete structure can be presented as a tree or in a multilevel browser, although as words (and potentially tasks) can belong to multiple groups it is not a tree in the data structure meaning of the word.

In a tree-based editor new instances are usually added to the tree using a context-sensitive popup menu. This menu contains a list of the object types that are possible to insert based on the selected item in the tree. The tree editor may in some cases not be the only editor to manipulate the structure. In Rational Rose it is, e.g., possible to create new classes directly in an open diagram, and the classes are then inserted as siblings of the diagram in the tree view.

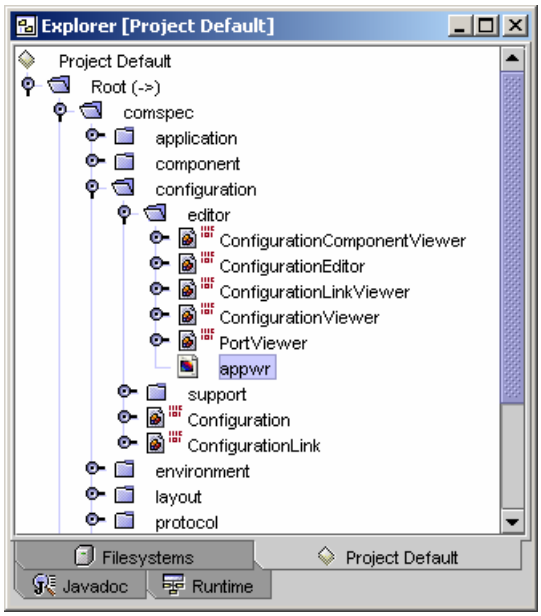

Figure 32 – Tree view from Forte for Java environment

In addition to defining the dataflow as presented under Section 6.3, the configuration editor of ComLink can also be viewed as a container. Configuration elements in ComLink can in theory play a role in the system at runtime without being connected to other elements of the configuration. Thus the configuration editor has a dual role, as editor for a container of non-visual objects and as editor for dataflow. In appearance the configuration editor is similar to the icon view of the file system used for Macintosh and Windows. When new instances are created from this kind of editor a palette is a natural choice for presenting the available object types. Instances are created similarly to how it is done in a layout editor, but the visual appearance of the configuration is only a help for the developer, and not visible to the end user. Hierarchical nesting in a configuration can be done similarly to how folders are used in a file system.

The structure editors make it possible to create instances of various classes, and arrange their structure and some of their relations. In addition there must be editors that allow editing of the various properties of the instances. The structure editors are also a natural point from which new user defined types can be created. A good example of this is Stagecast Creator where creating a new type is as simple as adding a new actor to the grid (adding new instances of an existing actor type is done by cloning an instance of the actor). The definition of the behaviour of the new type is done from the behaviour editor, while adding new properties can be done in the detail editor.

Table 6 - Summary of structure editors reuse potential

Part	Type of reuse	Dependencies	Notes
User interface editor	Component	-	E.g., ComLink layout, Hypercard card/background
Visual container for non-visual objects	Component	-	E.g., ComLink configuration
Table-based editor	Component	-	E.g., spreadsheet
Grid-based editor	Component	-	E.g., Stagecast Creator
Tree / browser editor with group support	Component	-	E.g., ComLink vocabulary, TASC task templates

6.5 Detail Editors

In existing environments, the properties of instances are usually edited either through a generic detail editor usable with any object type, or through specialised detail editors. Also the editing of some properties may be included in structure editors; size, position and colour of a visual object may, for instance, be modified in a user interface editor.

A typical example of a generic detail editor is the property sheet used in Visual Basic. This editor can display a list of the available properties of any object, together with simple editors for changing each property. The generic editor has the benefit that it is reusable. While this may be sufficient and suitable in most scenarios where professional software developers make a few changes to default values that are mostly appropriate, there are some problems:

- The list may contain many properties, creating a problem with overview. A general user interface guideline is that 5 +/- 2 items can be held in short term memory, and thus it is preferable if lists, menus etc. do not exceed this. Property sheets generally also do not have any way of grouping related properties.

- The small space for each editor may lead to sub-optimal interfaces. For example, it is not suitable for entering long texts, and value like URLs can usually not be viewed in full. In some cases dialog boxes are opened to edit a single property, but if used too frequently this disrupts the flow of work if entering a large amount of information.

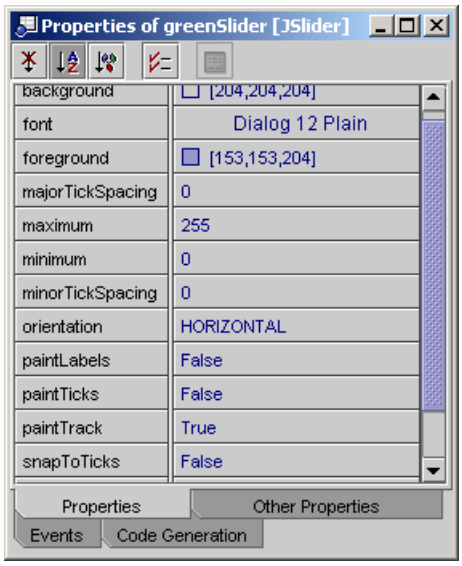

Figure 33 – Example screendump from a property editor

A better solution for the application developer can be to have a more specialised detail editor for each domain object. This was the approach followed for the objects editable by the facilitator role in Comspec. When a custom property editor is available for an object type (as, e.g., JavaBeans customizers) the editing of the properties is usually better organised and made easier for the application developer. The interface can be similar to a form fill-in or settings dialog box, and can be seen to support editing of the values of a set of properties grouped by related properties, and usually with a limited set of widgets to do the editing. This indicates that tool support could be used to simplify the task of creating such editors.

Table 7 - Summary of the detail editors reuse potential

Part	Type of reuse	Dependencies	Notes
Property sheet	Component	-	-
Custom detail editor	Conceptual	-	Good candidate for tool support

6.6 Debugging and Testing Tools

Many end-user programming environments have tools that help to correct or explain when the application under development is not working as intended. The scripting environments tend to use facilities similar to what is found in professional programming environments for the purpose. An example of this is Visual Basic for Applications, where the debugger allows the developer to step through the code one line/expression at a time, and step into and out of methods/functions. Breakpoints can be set at specific lines of the code, so when the code is executed it stops and allows the user to step through the code from the breakpoint. Also it is possible to watch the value of

expressions. The debugger is tightly integrated with the editor, e.g., for scrolling to and highlighting the current line that is executed.

In PBE approaches debugging may take the form of creating a “counter-example” or a correcting example. As an example, in Gamut debugging is done by pressing the “stop that” button when unwanted behaviour is performed, and giving a new example of how the application should behave instead.

Stagecast Creator’s debugging facilities is mainly focused on explaining why a rule does not fire. There is a single-step feature that allows the user to perform one time-step of the simulation. If the rule window for one of the characters is open during the single-step, the rule that triggers (if any) is marked. If the user wants to know why a certain rule could not trigger, the detail window of the rule can be opened. From this window a test-button allows the user to find out why the rule does not fire. It highlights any cell in the before-picture of the rule that do not match the current situation, or any other precondition that do not match. The debugging facilities are closely integrated with the rule editor, and there are also ties to the grid visual formalism.

In the dataflow-based environment Sanscript the debugging features are quite close to what is found in scripting systems. It is possible to step through a diagram, and at each step the current element is highlighted. Functions can also be stepped into, and this opens the dataflow diagram for the function. It is also possible to set breakpoints before and after functions, at which the normal execution of the program will pause and allow stepping. The values of outlets in the diagrams can be examined during stepping.

To summarise, with exception of PBE environments, the debugging and testing facilities of the existing environments are usually tightly integrated with the behaviour editors which makes reuse across behaviour editors difficult. On the other hand the concepts used for debugging is similar in most approaches: breakpoints are used for stopping the flow, inspection of variable values is allowed during execution, and single-step execution is used to examine the program flow.

Table 8 - Summary of debugging and testing tools reuse potential

Part	Type of reuse	Dependencies	Notes
Debugger	Component or source code	Tight integration with behaviour editors	Component require extensibility mechanism to support wide range of reuses

6.7 Component Management

Some of the end-user programming environments are extensible with new object types or functions. For instance, Hypercard can be extended with XCMD’s providing new functions that may be called from the scripting language. Another example is Visual Basic which allows new object types to be added through binary components. The support of most environments in this area is usually quite simple, and not suited for reuse.

Good ideas for reuse of extensibility support can instead be found in Sun's professional development environment Forte for Java. The "Update Center" of this environment (see Figure 34) is a good example of how to find, download and install components. In Forte this is used to manage the components of the development environment (e.g., editors for XML and Java, tools for version control and debugging etc.), and not the set of components used as part of the application that is built. Although made for professional developers, a similar approach could be useful to the application developer. One of the reasons the approach is limited to the development environment in Forte is probably that the potential number of JavaBeans to use as part of the application could be overwhelming. For a domain composition environment the number of available components will probably be much more limited, and as they are made specifically for the domain there is a greater chance that each component is relevant.

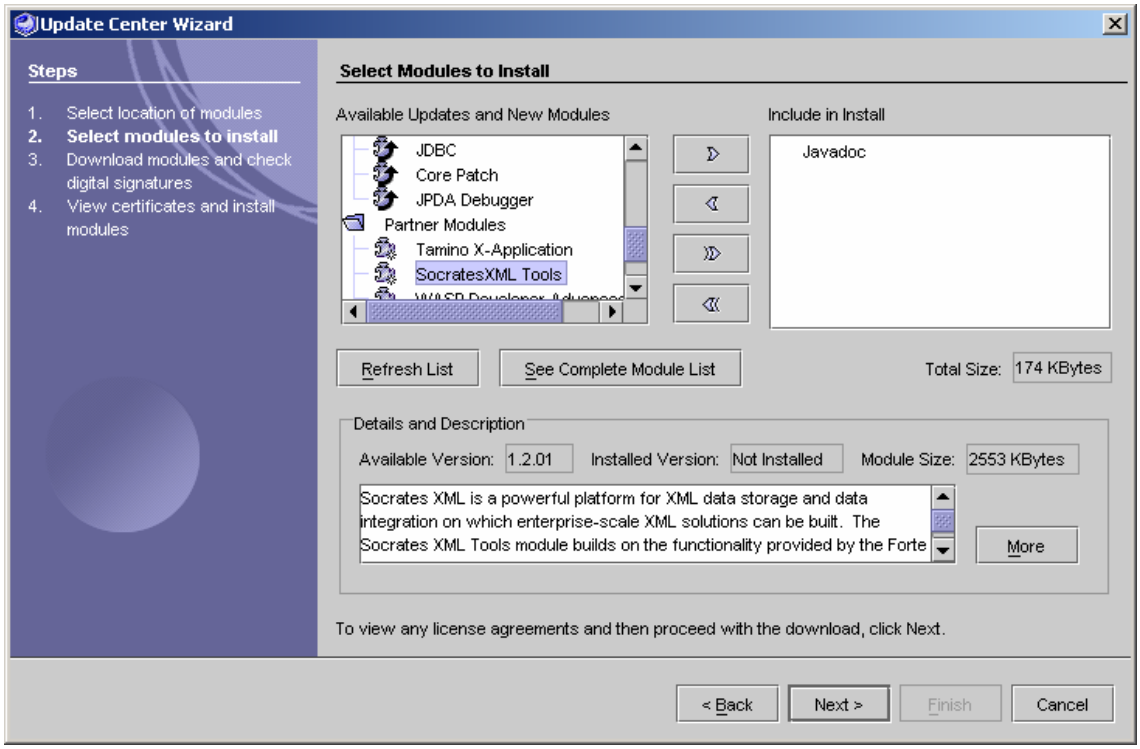

Figure 34 – Update Centre Wizard for the Forte for Java environment

Table 9 - Summary of component management reuse potential

Part	Type of reuse	Dependencies	Notes
Component manager	Component	-	Support download and install of environment specific components

6.8 Packaging and Deployment

Most of the end user programming environments now include some kind of support for packaging and deployment of stand-alone versions of applications. Early examples of environments would require that you had the tool, or at least a runtime version of it available. An example here is HyperCard, which requires either HyperCard or HyperCard player installed. In a HyperCard stack, resources such as pictures are typically included in the stack, and deployment of the stack on another computer is usually as simple as copying the stack to it (provided HyperCard or the player is installed).

Deployment for the web is also supported in some of the environments. In Stagecast Creator a simulation can be published to the web by selection a publish option from the file menu. This saves the simulation file together with a web page which can be used to access it. This web page includes a Java Applet that checks that the necessary player plug-in for Stagecast simulations is installed in the browser, and allows downloading and installing it if it is not. Similarly, ToonTalk can also translate an application into a Java Applet, and generate a web page for it.

Packaging and deployment is usually presented as a feature like “publish”, “save as…” or “transform” in the current environments, but is not otherwise easily identifiable as a component. Seen from a more abstract viewpoint, some common steps in the packaging and deployment process can be identified, like selecting a target file or directory for the deployable result, determining and including the required set of resources (e.g. pictures and sounds), and including the required runtime environment. Each of these steps may also allow adjustment of some options, which suggest that a stepwise wizard guiding the process could be a suitable user interface. As the set of options to can be related to runtime environment specific issues, and additional steps may be required in the process, a packaging and deployment component would need some extensibility mechanism of its own to support a wide range of reuses.

Table 10 - Summary of packaging and deployment reuse potential

Part	Type of reuse	Dependencies	Notes
Packaging wizard	Component or source code	-	Component require extensibility mechanism to support wide range of reuses

6.9 Reuse and Integration across Components

In this chapter, we have so far assessed reusable properties of existing environments, primarily by identifying components within different areas of concern. In this subchapter we discuss reuse possibilities across some of the identified components.

When comparing the identified components, the main potential for reuse across components seems to be for creating editors which have a central element of visual composition. These editors can be seen as specialisations and extensions of a generic object-oriented drawing editor. From the behaviour editors, this is the case for the data flow and flowchart editors, while from the structure editors, the user inter face editor and the visual container for non-visual objects are the clearest candidates. In addition the table- and grid-based editors to some degree can be regarded as special cases of visual composition.

Despite the similarities between these editors, there are also unique aspects of each editor. This suggests that a single, generic component is not likely to cover the needs for all these components, but rather that a framework supporting the common visual composition operations could be a good basis for reuse across the components. Recently, frameworks which could be suitable for this task have emerged in the Eclipse community (see Section 9.2.2).

Development of a full visual composition framework is seen as outside the scope of this thesis. Instead, we will here focus on another reuse aspect by raising the question of how to integrate multiple of the identified components to build an application composition environment. Integration of editors is an important issue because the different areas of concern described in this chapter need to be covered in an environment, but also because interesting environments can be built by combining multiple editors from the same area of concern; e.g. multiple structure editors. While most of the identified components are used during editing of an application, some will also be part of the runtime environment for the composed applications. Thus, integration is needed both during editing and at runtime. This challenge is addressed in the next chapter, by proposing an overall architecture along with a foundation framework and an editing framework.

6.10 Summary

This chapter has assessed reuse properties of the end user programming environments and environments from the case studies presented earlier in this thesis. The assessment was divided into different areas of concern, and each of these areas of concern was presented in a subchapter. A set of reusable parts have been identified, and was summarised in tables in each subchapter. The identified parts constitute a draft for a reusable part catalogue for building extensible application composition environments

In the last subchapter, reuse and integration across components were briefly discussed. A visual composition framework providing object-oriented drawing facilities was identified as the main candidate for reuse across parts, and could be a good foundation for many of the identified behaviour and structure editors. The further focus for this work will however be on how to integrate different editors in a single composition environment. This issue is addressed in the next chapter.

Chapter 7 Architecture of Foundation and Editing Frameworks

In this chapter an overall architecture for creating domain oriented application composition environments is outlined, and two component frameworks are proposed and outlined: a Foundation Framework defining basic mechanisms and services, and an Editing Framework defining how a set of editor components collaborate in a composition environment. This chapter directly relates to the research question: *What kind of software architecture is suitable to build such environments?* In addition, the two proposed component frameworks define rules of behaviour for components which enable extensibility of composition environments built from them, and thus is a proposed answer to the research question: *How can reusable parts be integrated in new environments?*

The work presented in this chapter is as a result from executing the use cases *develop overall architecture* and *develop component frameworks* of the tool framework architect, as described in Section 5.6.

The first section of this chapter gives an overview of the overall architecture, consisting of a layered model for how the different frameworks and components build on each other. This is followed by a short section on the use of platform independent descriptions and the use of Java/JavaBeans as an example of how the frameworks can be mapped to a target platform.

The Foundation Framework, which especially draws on the experience from the case studies, is presented next. This component framework supplements the selected platform with support for building document-based end-user programming environments by specifying additional services and rules for components. This framework is further used as the foundation for the Editing Framework, and is also the basis for domain specific component frameworks. The description of the framework is first presented in a platform independent manner, and is followed by a mapping to Java/JavaBeans in the next section.

Following the description of the Foundation Framework is a presentation and description of the Editing Framework. This framework defines the rules for how a set of editors can collaborate in an application composition environment. This work is based

on experience from the case studies and on the analysis of reusable parts of end-user programming environments from the previous chapter. Also for the Editing Framework, a platform independent description is provided before a separate section discusses mapping to Java/JavaBeans.

The description provided here is a proposed outline for the architecture of the two component frameworks. The intention is to describe the main mechanisms and rules of behaviour for components which should be included in the frameworks, and the work is based on generalisation of the experience from the case studies and studies of other existing environments. For each of the frameworks, the rationale for their design and the relation to previous chapters are described in the last subchapter of their platform independent description. The descriptions provided here should be a good starting point for future work on a more detailed architectural description or detailed design.

7.1 Outline of Overall Architecture

To be suitable for the development of extensible application composition environments, the overall architecture should support the identified roles from Chapter 5 in their tasks, and support the development and integration of the different kinds of components identified in Chapter 6. In more detail, a suitable architecture should fulfil the following criteria:

- support the identified developer roles by providing different layers of abstraction suitable to the division of labour;
- address both runtime and editing issues, supporting extensibility in both cases;
- support integration of different kinds of runtime and editing components, from the same and across the different areas of concern from Chapter 6.

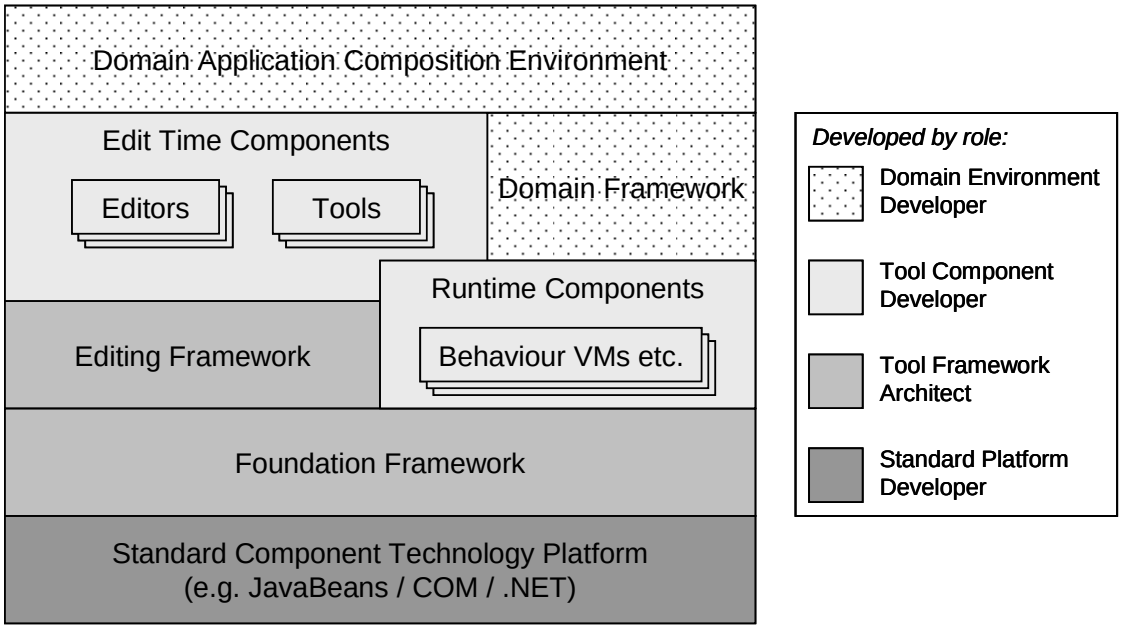

Figure 35 – Non-strict layer model of frameworks, components and environment

An overview of how the different frameworks, components and composition environments build on each other is presented in Figure 35. As the figure shows, the constituent parts can be seen as a layered architecture. The overall architecture is not strictly layered though, as each layer may access each of the below layers and not only the layer(s) immediately below. The Foundation Framework consists of mechanisms and components supplementing the standard component technology platform with additional features required for supporting document-based end-user programming. Components supporting runtime behaviour representation and execution, as discussed in Section 6.2, are built on top of the Foundation Framework, and these are along with the framework the basis for building domain specific component frameworks. The Editing Framework is also based on the Foundation Framework, and defines how different edit time components collaborate when used together in a domain application composition environment. Edit time components cover components in most of the categories defined in Chapter 6 and can include editors for behaviour, structure, and detail, and tools for language translation, debugging and testing, component management, and packaging and deployment.

The layered model of the overall architecture also helps providing different layers of abstraction for the developer roles. Tool component developers will use the Foundation Framework as the basis for runtime components, and also the Editing Framework when building components for the editing environment. Building tool components requires a good understanding of the frameworks.

The domain environment developer uses the foundation framework to build the domain framework. Depending on the available editors and tools, this developer may also need to use the editing framework and foundation framework to build the environment, but editor and tool components may hide this to a certain degree. The domain component developer operates at a similar level of abstraction, but has the existing domain framework and composition environment as a further guideline for the work. Tool support for the domain environment developer and domain component developer roles may further raise the abstraction level, and to a larger degree hide the Foundation and Editing Frameworks for these developers. Examples of such tool support are provided in Chapter 8.

The application developer role will use the abstractions provided by the domain application composition environment, and need no knowledge of the foundation and editing frameworks.

As presented in Figure 35, the overall architecture identifies points of extensibility for runtime and edit time components. The details of the extensibility support are defined in the Foundation and Editing Frameworks. The Foundation Framework also provides the mechanisms for extensibility which can be used to support extensibility in the domain framework.

7.2 Platform Independence and Platform Mapping

The component frameworks presented in this chapter are first described in a platform independent manner. By keeping the initial description platform independent, it is easier to later map the architecture to different platforms. Also, by not including platform

specific information, it can be easier to maintain an architectural level of the description and not introducing design details too early.

There are also some disadvantages of not utilising the selected platform in the initial architectural work. When the target platform is considered from the beginning, it is easier to draw on the strengths of the platform, and the mapping to this target platform can be simplified (but on the expense of ease of mapping to other platforms).

To illustrate how the platform independent description can be realised on a standard component technology platform, a discussion is also provided about how to map the architecture to Java/JavaBeans. The selection of Java/JavaBeans for this purpose was made partly because of Java's platform independence, and partly because the two case studies were made in Java.

Although the intention has been to keep the initial descriptions of the frameworks platform independent, it should be noted that many of the mechanisms and solutions included are motivated by experience from the case studies that were developed using Java/JavaBeans. This experience has to some degree affected the choice and style of solutions, and it is thus expected that mapping of the framework to Java/JavaBeans is somewhat simpler than mapping to other platforms. Still, mapping of the frameworks to other standard platforms such as COM or .NET would be possible, and probably requires an effort comparable to the mapping to Java for each platform. Assessment of further mappings is beyond the scope of this thesis.

7.3 Foundation Framework: Platform Independent Description

The goal of the Foundation Framework is to define and support mechanisms and rules of behaviour that allow coexistence and collaboration for the higher level components and component frameworks of an extensible domain oriented application composition environment, including the editing framework and domain frameworks. The selection and definition of mechanisms in this framework is based on input from the task descriptions of Chapter 5, the case studies in Chapter 4 and the reuse assessment of Chapter 6. Section 7.3.10 gives a further summary of the of the Foundation Framework design and the relation to this input.

As well-known mechanisms and design patterns have been reused and applied in the framework, the framework is likely to have some of the functionality in common with, e.g., more general object oriented frameworks, and such frameworks could be used a basis for the implementation of the Foundation Framework. However, support for the identified mechanisms is not always included in the standard component models, or their use may be optional.

This section starts by giving an overview of the mechanisms included in the Foundation Framework, and then describes patterns used in the design of some of the mechanisms of the framework, before a more detailed description of each mechanism is provided. A mapping to how it can be handled on the selected platform, which in this case is Java/JavaBeans, follows in the next section.

The following mechanisms are handled by the Foundation Framework:

- *Hierarchical context and service management*. Organises the main object instances in a hierarchical system that can be traversed and monitored. The context also manages services, and allows objects contained in the hierarchy to find and use these services.

- *Component repository*. The component repository defines a common way of keeping track of which components and object types are available, and giving the rest of the system access to this information. This can be used to create the selection of object types for palettes etc.

- *Object repository*. An object repository defines how existing object instances are organised and retrieved based on specified selectors, and also ensures that registered objects have unique IDs. Some of the background for this mechanism was discussed in the description of the mechanisms used in ComLink.

- *Persistence and data transfer*. Solutions are needed for storing the object structure which forms the application created by the application developer. In addition, persistence is also needed for the end user application.

- *Initialisation and coordination*. Allows objects to register for notification about major state changes in the application. This mechanism is used to determine when it is safe to start and stop performing runtime activities, locate other object to collaborate with, etc.

- *User level*. Keeps track of the current user level for the application or environment, and allows interested objects to be notified when it changes (e.g., to adjust the user interface to suit the new user level).

- *Domain objects and user-defined types*. Defines how user-defined types can be created by specialising domain object types, and how user-defined properties can be added to these types.

- *User-defined behaviour*. Defines how user-defined behaviour can be attached to the objects of the application, and how it is triggered.

7.3.1 Applied Patterns

This subsection describes patterns that have been applied in the design of some of the mechanisms in the Foundation Framework.

Change monitoring pattern

A problem that frequently occurs in software design is that some object needs to be informed when changes occur in another object, while at the same time a strong coupling from the source object to the observer should be avoided. An editor may, e.g., need to observe the changes to a domain entity, but the domain entity should not know anything about the editor. A well-tested solution for this problem is to apply the observer pattern [Gamma 1994] or a variant thereof. The Foundation Framework adopts the pattern variant used as the JavaBeans event mechanism [Hamilton 1997].

The pattern allows a source to be monitored for changes by multiple interested parties. The monitored source does not have any knowledge of each monitoring party, and requires only that registered listeners support a specific change listener interface. Figure 36 shows an UML class diagram for the change monitoring pattern. When a change occurs in the monitored source that can be of interest to external parties, it will notify each registered listener by sending an appropriate change message. To provide extra information about the change, a change event object can be sent with the change messages. In situations where a monitored source has different categories of change, it may allow monitoring of each category of change independently through separate change listener interfaces (and associated change events), and the source will then have an add- and remove-method for each category.

The pattern is used in the mechanisms defined in the Foundation Framework, but should also be applied when suitable in domain frameworks and solutions based on the Foundation Framework. In further descriptions found in this chapter where the pattern has been applied, the change events have been omitted from figures and description to avoid too much detail, but they should be used in implementations of the framework.

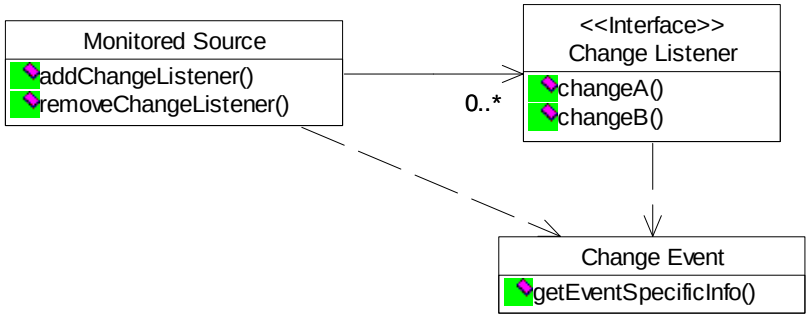

Figure 36 - Pattern for monitoring changes

Repository pattern

The Foundation Framework contains a set of repositories which are used to organise and find different kinds of item. While the different repositories are conceptually similar, there are also differences between them which would make reuse of a generic repository cumbersome. Instead, the repositories are based on a common pattern which is presented in Figure 37.

Each repository defines a set of operations for finding the items they organise. To support customised and extensible selection criteria, the find operations allows the client to specify any object implementing the Repository Item Selector interface to be used for determining if an item matches the selection criteria. Many repositories maintain additional information for the items they organise (meta-data), e.g., a set of properties which can be used by the selectors, and these are kept in repository item descriptors. The repository pattern also allows interested repository change listeners to monitor changes to the repository, and this is a reuse of the change monitoring pattern described earlier with the repository as the monitored source.

The repository may also provide a set of operations for changing its content, but these operations may vary more radically between different kinds of repositories. Some

repositories are built up by their clients at runtime, while others are more static and will, for example, set up their content when the system is started based on a XML-based configuration file or by scanning the content of a directory in the file system. Change operations are thus not included in this pattern.

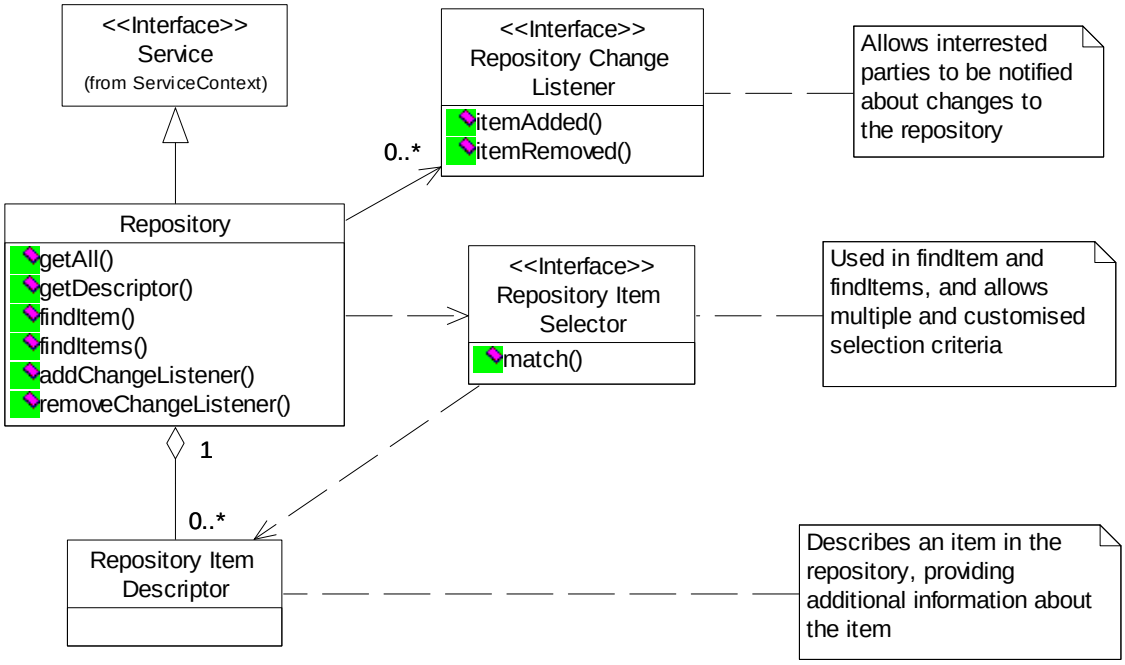

Figure 37 – Pattern for repositories in the Foundation Framework

A repository is a kind of service, and should be included as services in the hierarchical context and service management mechanism described later in this chapter.

7.3.2 Hierarchical Context and Service Management

Some kind of hierarchical organisation is necessary in most software systems. A well known design for this is the Composite pattern: "*Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly*" [Gamma 1994]. By having a well defined way of setting up and accessing this hierarchy, the component frameworks and components can utilise this common structure to traverse the hierarchy and monitor changes. Through the hierarchical context, it should also be possible to find and register services that are available. The design of the hierarchical context mechanism is mainly based on the experiences from the ComLink software where a simple containment and service mechanism was developed inspired by early drafts of the Java BeanContext specification [Cable 1998].

Figure 38 shows a UML class diagram of the context and service handling. Central to the context handling is the Context which can contain zero or more Context Child as children. As Context also extends the Context Child, a context hierarchy can be formed. To allow changes to the context to be monitored (e.g., by a view in the user interface) the change monitoring pattern has been applied. Interested parties implementing the Context Change Listener interface can register their interest in changes, and will receive notifications when children are added or removed from the context.

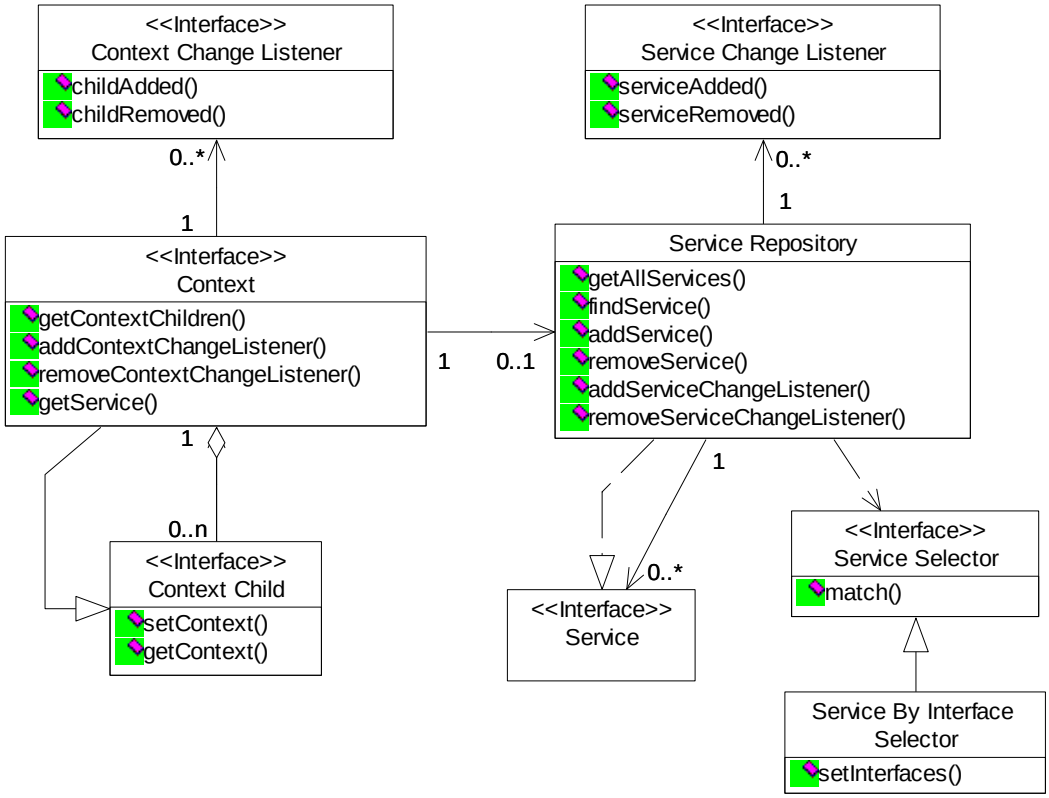

Figure 38 - Context and service handling

Each level in the Context hierarchy may have a set of associated services which is managed by a Service Repository. The Service Repository is based on the repository pattern described earlier. When a Context gets a request for a service, it will first check if it has the requested service in its Service Repository (if it has any). If not found locally, the Context will pass the request on to its parent Context. Frequently most services will be associated with the root context, and thus be available for all objects in the context hierarchy. The Service Repository also supports monitoring, and notifies each registered Service Change Listener when a service is added or removed. Most of the other mechanisms of the Foundation Framework rely on the context and service mechanisms to make their services available or to access other services.

7.3.3 Component Repository

The component repository component keeps track of which components are installed and which object types these components make available to the system. The design of the component repository is a variation of the repository pattern, and is show in Figure 39. As shown in the figure, the repository keeps track of two kinds of descriptors; for components and for object types. The main functionality of the repository is for finding and accessing object types, as this is the information most clients will need access to.

The object type descriptors allow a set of properties in the form of name and value pairs to be associated with the objects, and these can be used to organise the objects in groups or otherwise provide additional information about the object types. Object type selectors are defined which allow clients to find object types based on the interfaces they support or on the values of their associated property.

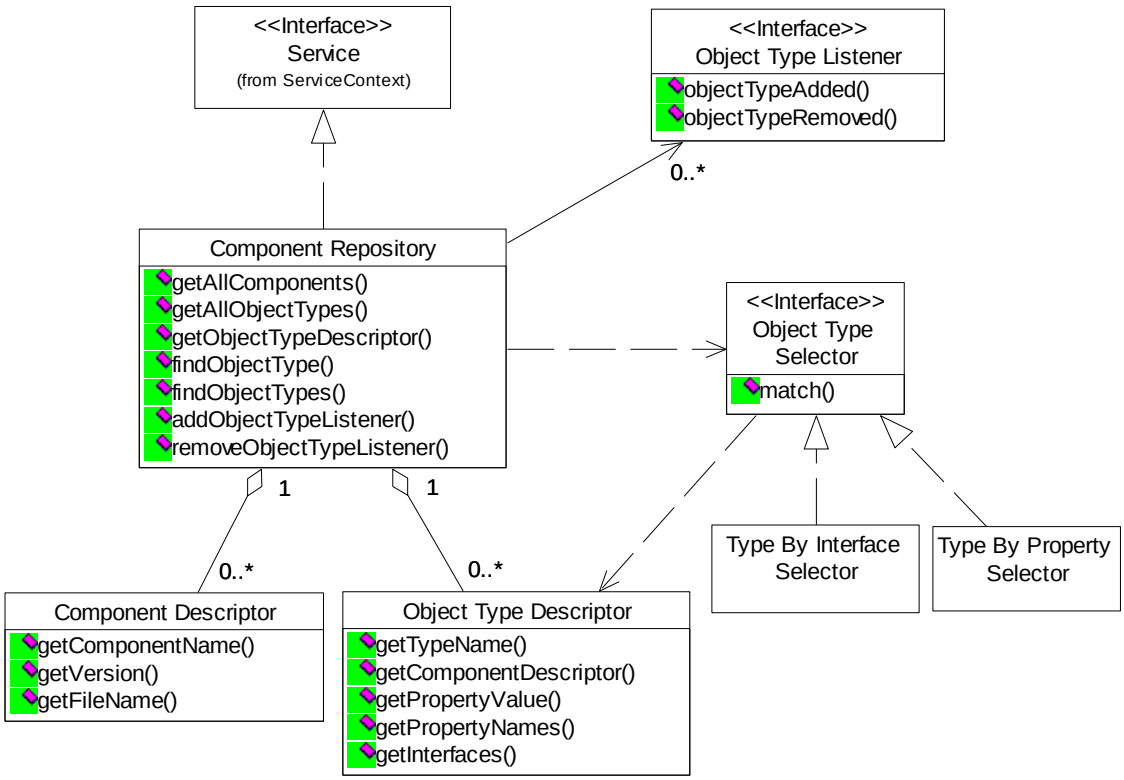

Figure 39 - Component Repository

One of the uses of the component repository is to keep track of which object types are available for the application developer to create instances from various points in the structure of the application. In this case, information such as which object types to include in a specific palette can be kept in the associated properties. The content of the component repository will typically be built at system startup, e.g., by scanning a predefined directory of the file system for all available components, and/or reading a set of configuration files. In systems built with the Foundation Framework, the component repository should always be made available as a service from the root context.

7.3.4 Object Repository

The object repository allows existing objects in an application to be organised and retrieved based on an extensible set of criteria. This, for instance, allows an editing system to find existing objects that are candidates for collaborations. Although a traversal of the hierarchical context could in theory be used for similar purposes, this repository makes common retrieval functions easier and quicker.

The design of the object repository of the Foundation Framework is based on the repository pattern combined with experience from the object registry of the ComLink software. Figure 40 shows an UML class diagram for the object repository and associated classes and interfaces. The repository manages and registers objects by unique names, and supports adding, removing and renaming of objects. To locate objects from the repository, selectors have been defined for retrieval by name, by the set of interfaces or class supported by the object, and by properties which can be associated with the object through the repository.

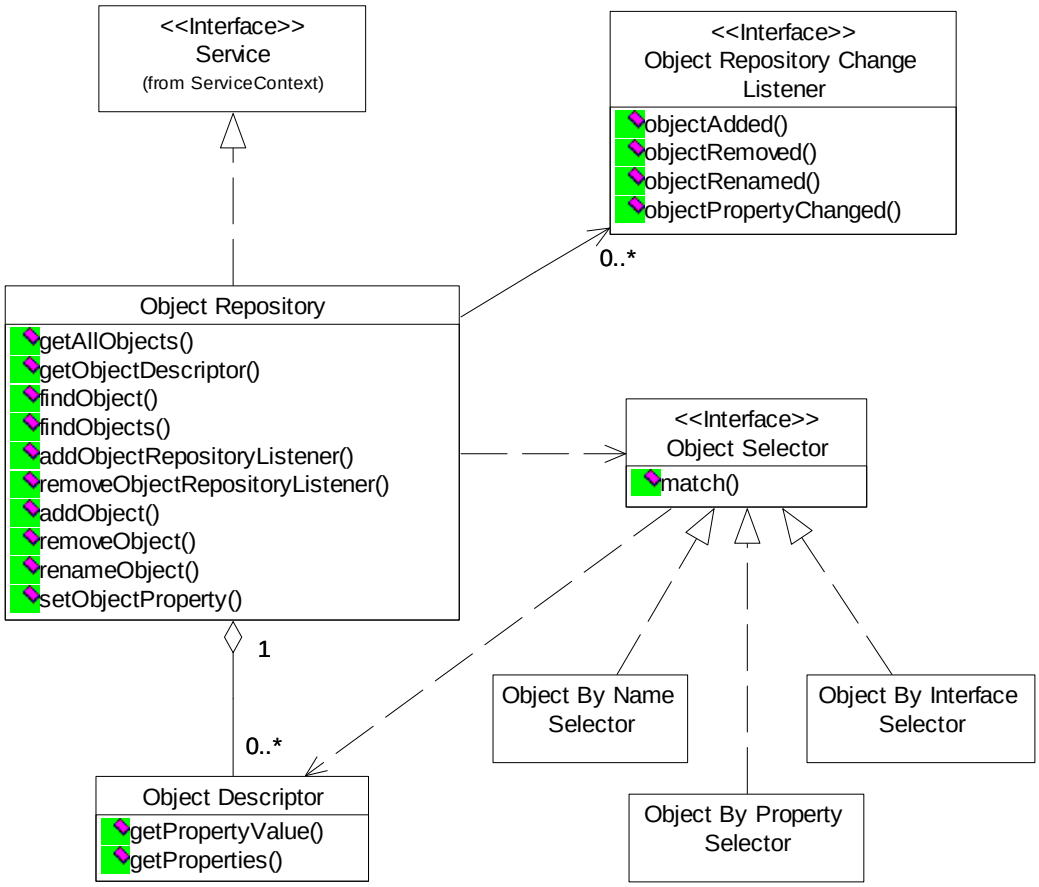

Figure 40 - Object repository

The names used for objects are strings, and an editing environment may allow the application developer to select the names of the objects as long as they are unique. The names allow a set of objects to be presented to the application developer when a selection is needed.

Future versions of the object repository might replace the support for management of names of objects by a more general mechanism that could ensure uniqueness of certain object properties within specified parts of the context hierarchy. An option for limiting the search to specified parts of the context hierarchy could also be considered.

7.3.5 Persistence and Data Transfer

As described in Section 5.2, document-based storage is used as the persistence mechanism in most end-user programming environments, and the persistence and data transfer mechanisms of the Foundation Framework are focused on supporting document-based storage and development. Both during development and at runtime the application is constituted by its object instance structures. At design time, the whole object structure which the application developer creates and edits is stored to and retrieved from document-based storage. Parts of the object structure can also be involved in data transfer as part of clipboard and drag & drop operations, or can be exported to or imported from separate documents to support copy & paste reuse across applications. Both for storage and other data transfer, the mechanism must support

persistence of references that constitute the containment hierarchy as well as cross-references between objects in this hierarchy.

In addition to storing the main object structure, meta-information about which components are used in the application must be stored. This information must be possible to read separately from the main object structure, to allow inspection of an application to determine which components are required. During storage, the required meta-information can be found using the component repository.

During runtime, the parts of the object structure that is modified and which should be restorable in later sessions of application use (e.g., game status in the role playing game example) can be stored and retrieved from document-based storage.

Preferably, the implementation required of each component to support persistence should be kept as small and simple as possible. As part of this, support for persistence of the most frequently used object types should be predefined. The mechanism should also be robust and allow the system to evolve, and the performance of the mechanism should be satisfactory.

Another issue is the format of the document files. In some cases (e.g., during development), it is preferable that a standard format is used, possibly readable and editable in a text editor. In other cases, it is important that the format is not easily readable, for instance, to avoid unwanted editing of a deployed application.

Some of these requirements are conflicting. In addition to the last point about file format, another possible conflict is between performance and ease of implementation. A reflection mechanism can be a good basis for creating a persistence solution where the component developer does only have to declare what is persistent and what is not, avoiding specific persistence handling code (e.g., read and write methods). However, reflection has a performance penalty, and may thus not be suitable in situations where high performance is required.

At design time, the persistence mechanism should store the whole object structure (or a substructure of it), including all non-transient properties of the objects. When restoring, all objects in the structure are created, and their properties' values are restored. During storage and retrieval, cross-references have to be handled differently than containment references. When an object being stored has containment references to other objects, these references and the contained objects should also be stored. However, parent objects are not stored as a consequence of storing a contained object, as this would always lead to storing the full hierarchy. When an object being stored has a cross-reference to another object, the referred object is not stored as a result of the cross-reference. However, if the cross-referred object is stored because it is part of the containment hierarchy of the stored substructure, the cross-reference will also be stored. If the persistence mechanism is used as part of a copy operation, all references will be kept local to the copied structure instead of referring to the original structure. Figure 41 shows an example of this, where the parent of e' is c' (not c), and the reference from e' leads to f' (not f). Also, the cross-reference from d to b is not kept in the copied substructure, as b is not part of the containment hierarchy of the substructure being copied.

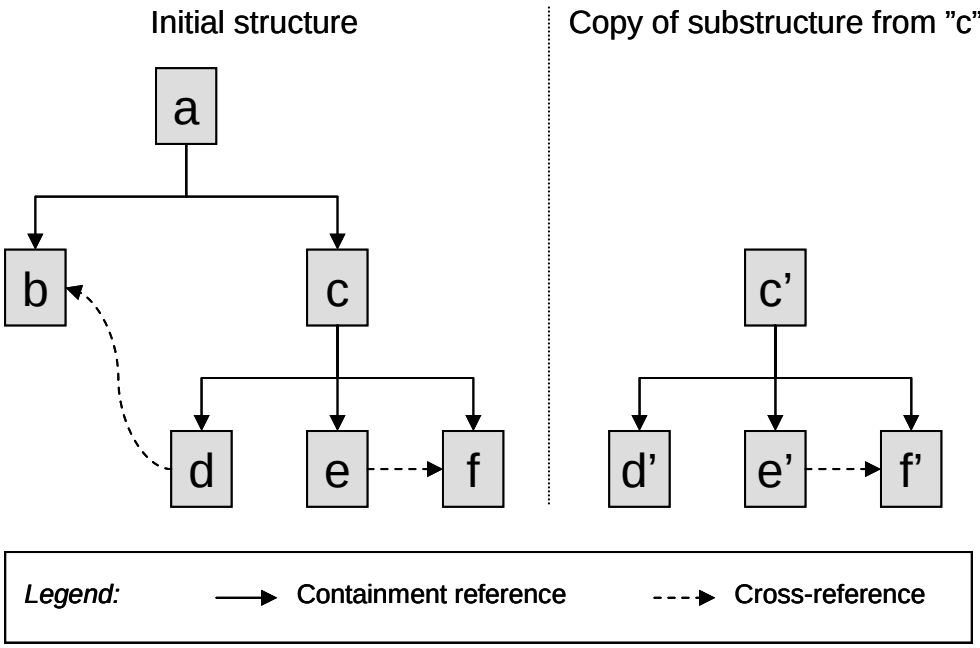

Figure 41 – Copying of structures including containment and cross-references

At runtime the requirements are somewhat different. At one extreme, a solution to runtime persistence is to use the same mechanism as for design time, which means saving the whole application. At the other extreme end is saving only the difference between the original design state of the application's object structure and the current state in the running application. Both of these extremes have their pros and cons. While saving the whole application is an easy solution as it does not require a special solution for runtime persistence, much more data is stored than is actually required. Also it prevents the use of old stored runtime state (e.g., saved games in the role playing game case) to be used with new versions of the application. Despite these problems this scheme has been used in products, for instance, in Hypercard stacks.

The other extreme solution requires that current runtime state of the application is compared to the original state when storing the runtime state. This is a more complex operation, and requires more processing than storing the whole application. There is a good chance that old stored runtime state will work with new versions of the application in this case although it can not be guaranteed (e.g., a bug fix which changes the initial value of a property that may change at runtime can already have been saved as part of runtime state). To store the difference between the original and current state such a mechanism must store all changed property values. In addition to changes to simple properties like integer and string value, this can include changes to substructures of the original object structure. New objects may have been added, and some of the original objects may have been removed. One way to do this is to follow an approach similar to what is done in the long term persistence scheme for JavaBeans, which allows a set of expressions to be stored. In this case, such expressions modify the original structure, e.g., by removing and adding objects, and setting other property values.

An alternative between these two extremes is to mark which properties of each class should be persistent at runtime. When storing the runtime state, the whole object structure is traversed, and all properties that are marked as runtime persistent will be stored. As for the previous alternative, this mechanism must support objects that has

been added or removed, but this time only when they are contained in collection properties that was marked as persistent at runtime.

The last alternative that will be assessed here is to require that the domain framework is designed in a way that locates all state that changes in a substructure of the application structure. Storing the runtime state could then be done in the same way as storing other substructures of the application. This solution can be appealing from a performance viewpoint, but can also lead to unappealing designs as runtime state of an object have to be split from the rest of the object.

From the discussion above it is clear that there is probably no document-based storage scheme that is ideal for all situations and all requirements. The Foundation Framework addresses this problem by not defining a single persistence scheme, but instead defines rules that ensure that enough meta-information is available that different persistence schemes can be implemented. These schemes can use introspection at runtime (as the Serialization and long term persistence for JavaBeans do), or can, e.g., generate code for each class based on introspection on the classes in the code generation phase. The persistence schemes can be developed independently of the domain frameworks, and can, for example, be provided by a tool component developer. The environment developer should select a persistence scheme suitable to the domain framework and environment that is built. In some situations an environment may support multiple persistence schemes. It may, e.g., be useful to have the application stored in an XML file during development, and transferred to another format when the application is packaged to avoid that application is changed by the end user. The role playing game case is an example where this could be preferred.

The Foundation Framework defines the following rules that support persistence:

- Properties of an object type used to contain cross-references have to be identifiable by from the meta-information for the object type, and thus allow the persistence schemes to distinguish between cross-references and references to owned objects.

- It must be possible to mark a property of an object type as being persistent at runtime, thus allowing a runtime state persistence mechanism to store only part of the structure.

- To allow inspection of which components are required for an application, the persistence-mechanism must store meta-information of used components, and this information must be possible to read separately from the main object structure

7.3.6 Initialisation and Coordination

Usually the objects constituting an application will perform the initialisation that is required when the object is created or restored from persistent storage, but in some cases, additional coordination is needed for timing of initialisation and other activities performed by the object. Some of the objects in an application may need to behave differently during design and runtime, or to perform some initial actions when they become an active part of the application, and similarly need to clean up when becoming

inactive. This can, e.g., involve registering for notifications with other parts of the system or starting a new thread of activity. In the ComLink case, this was done through a document state change notification, with a common global state for the application.

In the Foundation Framework, the coordination of activities is performed by allowing interested objects to monitor changes in the application status. As Figure 42 shows, the mechanism is an application of the change monitoring pattern. The Application Status Tracker can be an independent service, or be implemented, for instance, by the top level application, and will fire changes to registered listeners whenever the status of the application changes. The exact states the application can change between has been left for a future detailed design of the framework, but should at least include changes between initiating, editing and running states. In systems built with the Foundation Framework, there should always be an application status tracker registered as a service in the root context.

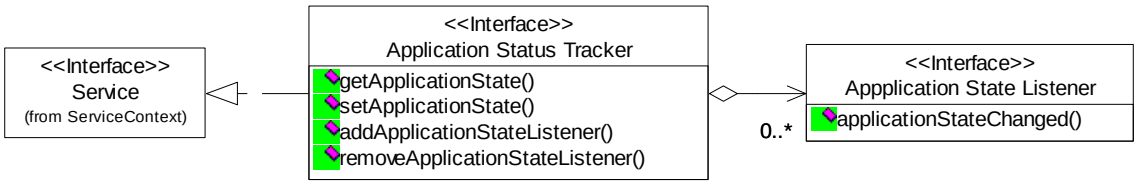

Figure 42- Initialisation and coordination

In some situations it would be useful to allow the domain framework and domain classes to decide which parts of a loaded application are active. In the ComLink case this could, e.g., have been used to allow multiple configurations to be present in an application, but with only one active at a time. To support this feature, a containing context can override the default application status tracker, and thereby control the application status for contained objects. Further, the application states should include suspend and resume states similar to those of the Component Configurator pattern [Schmidt 2000].

7.3.7 User Level

Both during editing of an application and at runtime there may be a need to differentiate between users at different levels of experience to adjust the user interface. In ComLink, this was used to separate between the integrator, facilitator and end user, and a similar approach has also been used in, e.g., HyperCard. Although the most obvious use of user levels is to differentiate an editing environment, it can also be used as runtime, allowing more gradual difference between editing and runtime. To support this, a user level tracking mechanism is included in the foundation framework.

The design of the user level handling mechanisms is presented in Figure 43. It parallels that of the initialisation and coordination by being a simple service following the change monitoring pattern. As for the initialisation and coordination mechanism, the definition of the state space for user levels have also been left for a future detailed design of the framework.

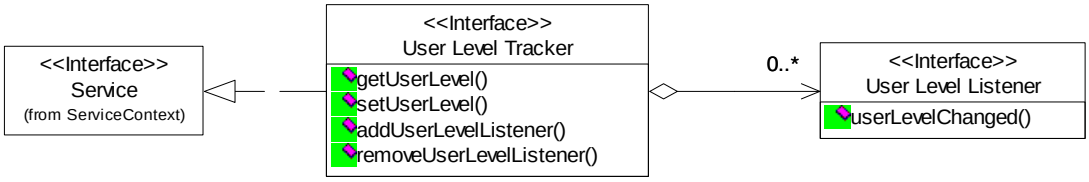

Figure 43- User level handling

7.3.8 Domain Objects and User-Defined Types and Properties

The Foundation Framework support for domain objects and user defined types provides part of the foundation for supporting the *edit application* use case of the application developer (from Section 5.2.2), including the detailing in the use cases for *create instances within structure*, *edit property values*, and *define types*. The mechanism is intentionally kept quite simple, but at the same time is similar to what is provided in successful end user programming environments (e.g., Stagecast Creator). A user-defined type is created based on a domain type that is designated to be a base type. There are two kinds of predefined properties associated with a user-defined type: shared properties and regular properties. A shared property has one value kept by the type that is shared by all instances (similar to a class variable) while regular properties are kept by each instance. The type holds a default value for each regular property that is used to initialise the value of new instances. When creating a user-defined type, the application developer edits values of shared properties and the initial values for regular properties. The application developer can also define a set of user-defined properties in addition to what is provided by the domain type. Behaviour can be associated with the user-defined type when the definition of the base type has properties for this.

Figure 44 shows the main collaborators in the in the user defined type mechanism. The Domain Entity represents the instances which the application developer creates. The domain entity instances are created based on one Domain Entity Type (usually mapped to a domain entity class in realisations) from the set of types defined by the domain framework. For each Domain Entity Type there is a Domain Entity Descriptor which provides addition meta-information about the type and the properties it defines, as well as default values for each property. Each domain entity instance has a set of properties as defined by the Domain Entity Type, and in addition a set of user defined properties as defined by the associated User Defined Type. The User Defined Type defines a set of user defined properties, and it also contains default value or shared value for these properties as well as for the properties defined by its associated Domain Entity Descriptor. The description of each property defined by the User Defined Type and Domain Entity Descriptor is maintained by a Property Descriptor.

Figure 45 shows an example of the user defined type system in use in the role playing game example. In the left part of Figure 45 a monster type is defined as a domain type. The right part of the figure shows how a domain entity descriptor is associated with the monster type, and how two user defined types have been derived from the monster domain type. Two instances have been created from the wild cat type, and one instance from the dragon type. All the instances are of the same monster domain type which determines their predefined behaviour and properties, while the association to their user defined type determines their user defined properties and behaviour.

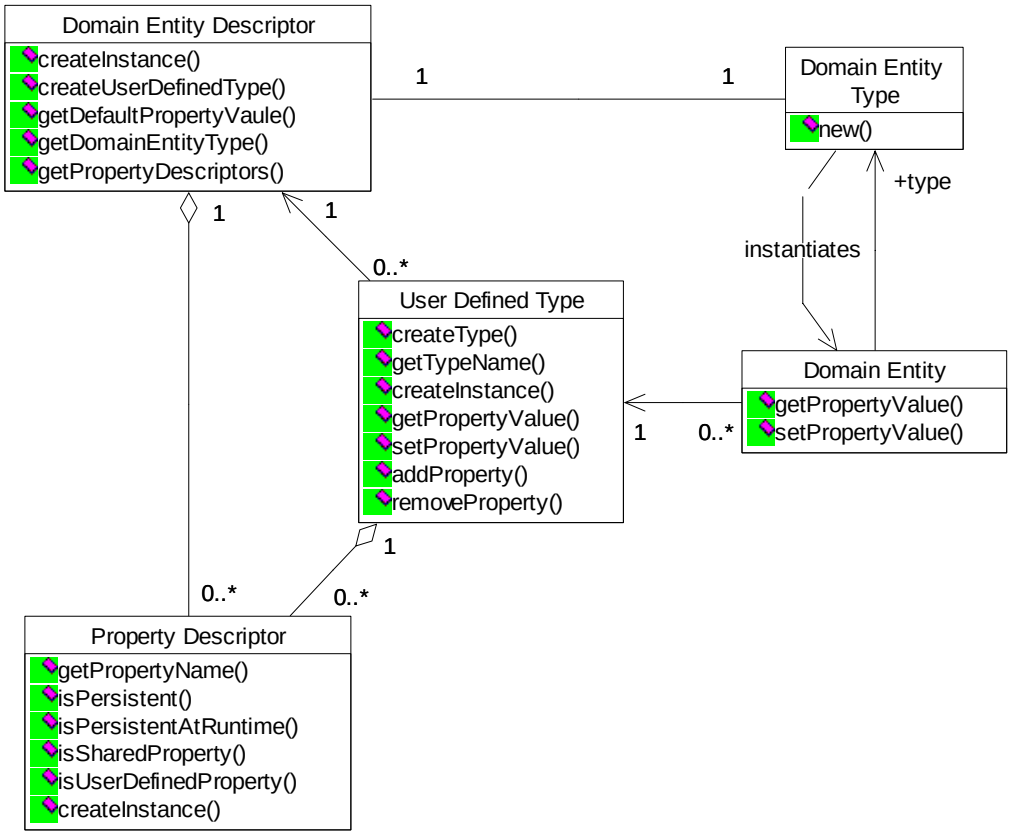

Figure 44 – Domain objects and user defined types

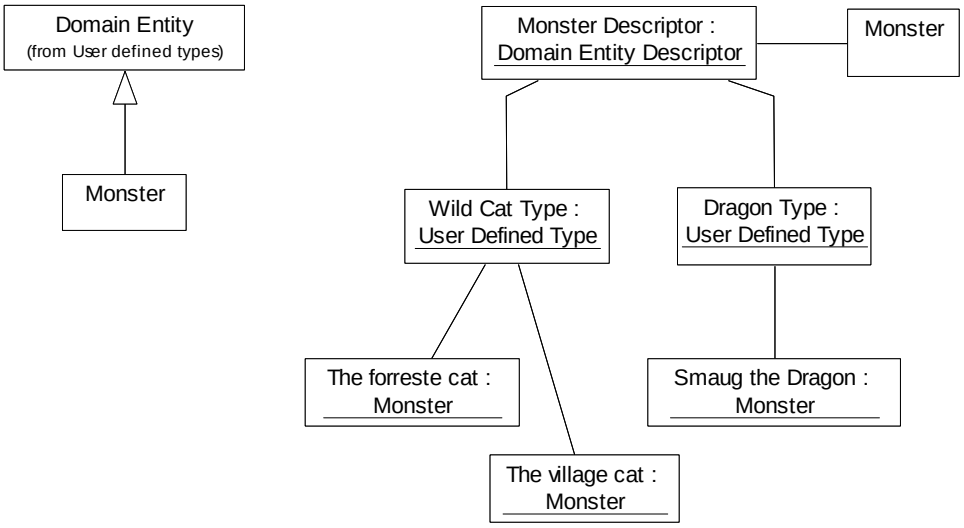

Figure 45 – User defined types based on a domain type (role playing game example)

When an instance of a user-defined type is created all regular properties are initiated with the default values from the type, and the instance is given a reference to its user defined type object (as shown in Figure 46). Regular properties are returned and set as normal in the selected component platform. When a shared property is requested the instance pass the call to the type, which returns the shared property value. The shared value can not be set through the instance and thus the instances do not have set-methods for these.

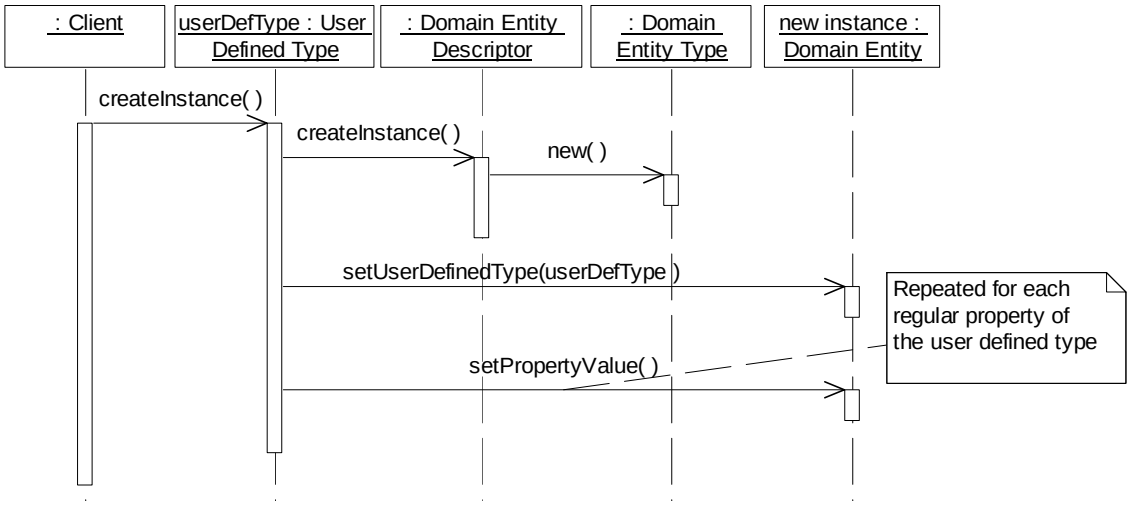

Figure 46 - Create instance from user defined type

Figure 47 shows the message sequence for creating a new user defined type. The sequence is initiated by a client request to the descriptor of the domain entity type which is used as the base type. The descriptor will create a new User Defined Type instance, and initiate itself with default values for all properties defined by the base type. After the type is created, the client can add user defined properties with associated default values, and for each of these a property descriptor will be created by the user defined type.

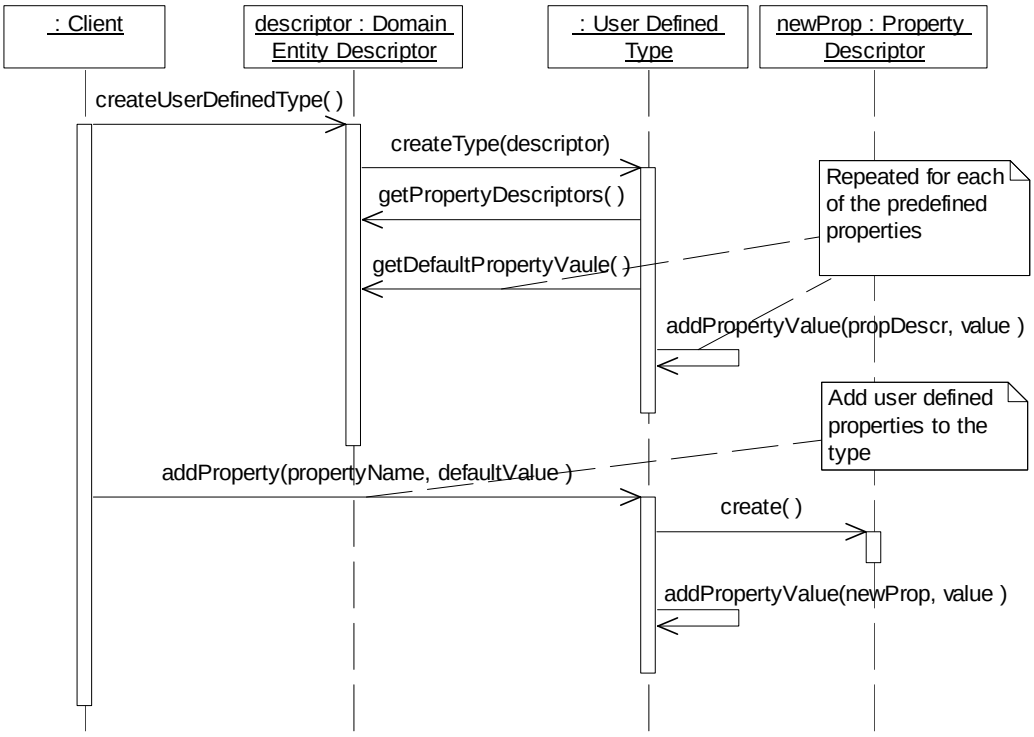

Figure 47 - Create user defined type

As shown in Figure 48, subtypes of the Property Descriptor are used by the framework to allow description of properties used to hold contained domain objects and cross references to other domain objects. This meta-information can be used by the persistence mechanism as well as, e.g., generic structure editors.

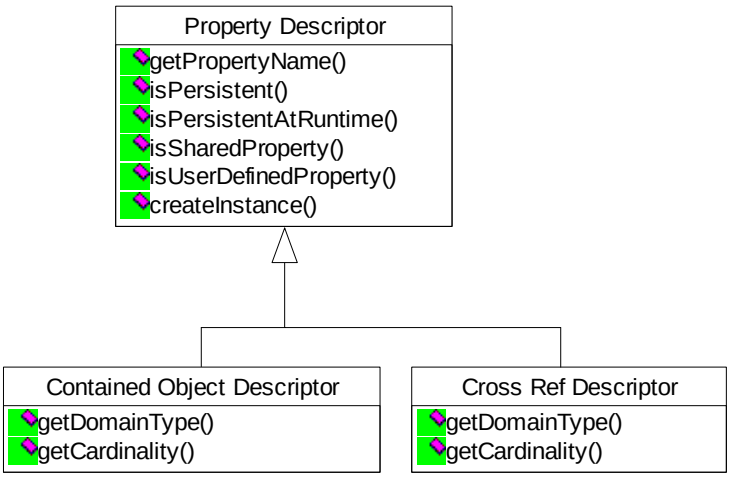

Figure 48 – Property subtypes

The hierarchical context and service management mechanism is used to organise the hierarchy of domain entities, and thus domain entities realise the Context Child (and Context if the entity can contain other entities) as defined in Section 7.3.2. As shown in Figure 49 the domain entity also defines listener interfaces for keeping track of changes to both user defined and predefined properties. Together with the listener interfaces defined by Context Child (and Context), this allows the user interface and other interested parties to keep track of all changes to both the containment hierarchy and individual properties of the entities.

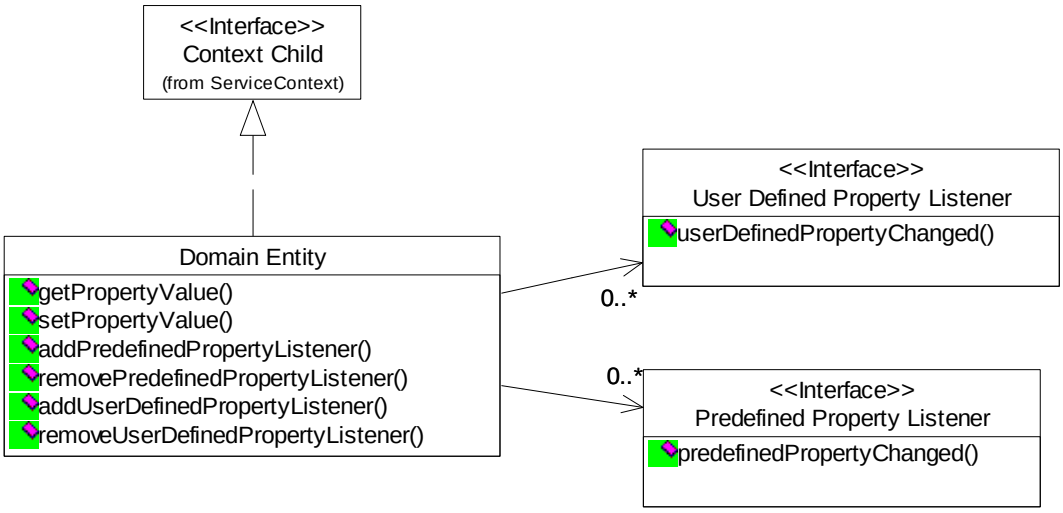

Figure 49 – Domain entity in context

7.3.9 User-Defined Behaviour

As covered in Section 6.2, many different approaches can be used for behaviour representation and execution in end-user programming systems, including imperative as well as rule or constrained based models. The Foundation Framework does not predefine a specific model for the user-defined behaviour, but leaves this choice open to the domain environment developer. When developing the domain framework, the environment developer will have to select one or more components to use which allows definition of user-defined behaviour. These components will be provided by tool component developers.

The environment developer will also decide where in the domain model the behaviour should be attached. Behaviour objects can, e.g., be attached as properties of domain objects and be performed under the control of the domain framework, or the behaviour can be defined centrally as a separate domain object and attach to events generated by other domain objects. In a rule-based system, the rules must be checked regularly, e.g., by a separate thread that triggers at regular intervals, or at specific point of execution decided by the domain framework. The most appropriate way of attaching and triggering the behaviour depends on the kind of end user programming technique and on the domain framework. This approach allows the domain framework to be in control of most of the behaviour of the system, while trigging the user defined behaviour when appropriate. The user-defined properties are usually not used directly by the behaviour of the domain framework, and it is up to the user-defined behaviour to make use of these properties.

Reflection allowing access to the methods and parameters of classes, as well as properties and events provided, can be used in the implementation of an end user programming language to interact with the domain objects of the application. This level of reflection is usually provided by the standard component platforms.

Components implementing a behaviour mechanism must ensure that the user-defined behaviour can be stored persistently as part of the application. User-defined behaviour will frequently have to refer domain objects. If the behaviour is stored persistently as an object structure, cross-reference objects should be used for references to domain objects to ensure correct storage and data transfer operations.

7.3.10 Rationale for the Design

This subsection gives a summary of the rationale for the selection and design of the mechanisms of the Foundation Framework. The task descriptions of Chapter 5, the case studies in Chapter 4, and the reuse assessment of Chapter 6 have been the main input to the design.

Among the task descriptions in Chapter 5, the tasks of the application developer have had most impact on the content and design of the Foundation Framework. Although the application developer will not directly interact with it, the composition environments built from the framework need a foundation for supporting the tasks of this role.

The *manage application versions* use-case specifies the use of documents as a suitable means, and the persistence and data transfer mechanism is thus based on document

based storage. The *package application* use-case is not covered by the framework, but this is considered to be a suitable task for a tool component. For the *test application* use-case, the framework provides the initialisation and coordination and the user level mechanisms, which can be used to differentiate the runtime behaviour for testing purposes. Both the context and domain object mechanisms also facilitates testing, by supporting traversal and monitoring of the application content, which can be used to build generic inspection tool components.

Support for the *manage components* use case is mostly left as a task for tool components. The component repository does however provide some support, both by being a central registry for finding installed components, and by providing information on which object types are associated with which components. The last information can, e.g., be used to implement support for the *remove component* detail use-case, which involves checking that no instances exists made from object types provided by a component to be removed. The persistence mechanism discusses how to support the *inspect which components are required by application* detail use-case.

Many of the mechanisms defined in the Foundation Framework contribute to support the *edit application* use-case and its subtasks. For *define types* and *edit property values* the domain objects and user-defined type mechanism provides essential support. The user defined behaviour section describes the frameworks approach to the *define behaviour* use-case. For *create instances within structure* multiple mechanisms contribute, including the hierarchical context (for structure), domain objects (for creating instances), and the object registry (for naming).

The selection and definition of mechanisms in this framework is also inspired by the experience from the case studies covered in a Chapter 4, and especially from the ComSpec case. The component repository, object repository, initialisation and coordination and user level mechanisms are all refined versions of mechanisms used in the case studies. The case studies also contributed with insights used in the description of the persistence mechanism.

From the discussion of components identified in Chapter 6, support for integration of multiple of the components in a single environment both at runtime and during editing was raised as central issue. Each of the mechanisms of the Foundation Framework can be said to contribute to supporting extensibility and integration, as they define a common ground for how components find, use and store content of the application, how they can find and use services provided by other components, and for how they are notified of important state changes.

7.4 Foundation Framework: Mapping to Java/JavaBeans

This section describes issues involved in mapping the platform independent model of the Foundation Framework to Java and JavaBeans.

7.4.1 Patterns

As described in the platform independent description, the Foundation Framework adopts the variant of the change monitoring pattern used as the JavaBeans event

mechanism. There is therefore a direct match between the change monitoring mechanism of the Foundation Framework and Java /JavaBeans.

The mapping of the repository pattern to Java is also straight forward. The items organised by the repository can be kept in an instance of one of the Java collection classes inside the repository, and the collection classes also provide manipulation functions that can be used to implement most change functions. As the repository uses the change monitoring pattern, the Java event mechanism will be used for notifying about repository changes.

7.4.2 Hierarchical Context and Service Management

The Java BeanContext specification "Extensible Runtime Containment and Services Protocol for JavaBeans" [Cable 1998] defines a hierarchical context and a service mechanism that is quite similar to the mechanism defined in the platform independent description. As the design of the hierarchical context mechanism is mainly based on the experiences from the ComLink software, which was in its turn inspired by an early draft of the BeanContext specification, it is natural that there is a close match.

In addition to the hierarchical context, the BeanContext specification also specifies a service mechanism. This mechanism has a richer set of features than the simple approach used in ComLink, and allows services to dynamically be added and removed from the context, as is also supported in the platform independent description. The support for services is not included in the basic BeanContext interface, but instead in the BeanContextServices interface that extends the BeanContext. As support for the services mechanism is required in the Foundation Framework, the BeanContextServices is a better match than the basic BeanContext. The specification of the BeanContextServices also defines when and how to initialise the connection to the services. When the context of a child is set, the child may ask for available services. If a service is not available at that time, the child can register to be notified when new services are made available.

The main difference between the design in the platform independent description and the BeanContext design is that BeanContextServices also take most of the responsibilities of the Service Repository. This is a deviation from the repository design pattern used in the Foundation Framework. There is thus a choice to make in the mapping on whether to most closely follow the platform independent design, or to use a standard of the realisation platform. To best facilitate reuse and integration with other JavaBeans, the best choice here is to adopt the BeanContext design. The BeanContext specification is optional in JavaBeans development. In systems built using the Foundation Framework, the context mechanism should always be used, and with Context being realised be BeanContextServices.

7.4.3 Component Repository

In JavaBeans components are packaged in deployable JAR files. These files are basically zip file archives with some reserved file and directory names. One file that can be included in a JAR file is a Manifest which is used to describe the content of the file. This can, e.g., be used to specify a main class to generate an executable JAR file, or to

specify which JavaBeans it contains. The Manifest can also contain additional information associated with the beans in the form of name and value pairs, and also the same mechanism can be used to describe the component as a whole.

A Java-based implementation of the component repository should utilise the information from the Manifest to build its content during system startup. One approach could be to scan a predefined directory for all available JAR files, and go through the Manifest of each of these.

7.4.4 Object Registry

The mapping of the object registry to Java follows the mapping for the repository pattern.

7.4.5 Persistence and Data Transfer

The target platform (Java/JavaBeans) has two persistence solutions that should be examined. All JavaBeans are required to support the Serialization. This mechanism allows a graph of objects to be written to a stream, and can be used both for document-based persistence and for data transfer operations (for instance, copy / paste). Serialization is based on introspection on the objects involved, and by default will write all variables defined in an object that is not marked as transient. Classes can partly override the mechanism by implementing read and write methods, or can override it more fundamentally by implementing the Externalizable interface. A problem with Serialization is that updates to classes may break the compatibility with previously stored files. A new Java version is, e.g., likely to break compatibility for some of the classes stored with previous versions, and the documentation classes in the Java Swing packages include notes that warn about probable Serialization incompatibility with future versions.

The other persistence mechanism for JavaBeans is the long-term persistence (included from version 1.4 of Java). As the name tells, this mechanism is meant to be more robust for evolving systems and the document is stored in XML. Similarly to Serialization the mechanism allows a graph of objects to be written to the stream, but instead of using the variables of the objects, the default mechanism is based on the JavaBeans properties of the objects. Properties can be marked as transient using the PropertyDescriptor of the BeanInfo. The default mechanism can be replaced for a class by using a custom persistence delegate for the class.

A problem that is common among these two persistence schemes is that they do not support exporting parts of an object structure by default. If there are cross-references to objects that are not part of this substructure, these will also, by default, be stored by these persistence schemes. A simple reason for this is that there is no special kind of Java object references to use for these cross-references. From the 1.2 version of Java, the java.lang.ref package is included with support for reference objects that allow better control of the garbage collection. The SoftReference and WeakReference classes are similar to what is needed, but support for the persistence mechanisms is not included.

In the ComLink software, the Serialization mechanism was used for clipboard operations, while a custom mechanism requiring read- and write-methods for each class

was used for document-based persistence. To deal with the cross-references, a special reference class was created, and the Serialization mechanism was modified to deal with instances of this class. Whenever such a reference was encountered, an ID was generated and written to the stream instead of the reference. Also, a list was kept of all objects written to the stream (through normal references), and after all objects were written, a table was written which recorded the association between IDs and objects that was actually written. When a file was read the cross-reference objects read the recorded ID and registered for a reference resolve phase. The resolve phase occurred after all objects had been read, and restored real object references based on IDs when the referred object was part of the archive, or set the reference to null otherwise.

While the ComLink approach works, it requires some additional code for each property used hold a cross-reference. An approach that avoids this problem is to use the available context hierarchy information along with the object type meta-information to determine whether the property is a cross-reference, and whether the reference leads to another object contained in the sub-tree being written.

The Java mapping for the Foundation Framework thus adds the following rules for supporting persistence:

- As the framework is built on top of JavaBeans, classes with persistent instances must implement the Serialization interface.

- The default serialization and long term persistence mechanisms are enhanced by subclassing java.io.ObjectOutputStream and java.beans.XMLEncoder to use the context hierarchy and object-type meta-information to determine whether to store cross-references to objects.

Following the above rules also allows the two persistence mechanisms available in Java to be used to create default mechanisms for environments developed with the Foundation Framework. Support for serialization is a requirement, and when enhanced to support cross-references it can be used both for data transfer (for which it is a good solution) and for persistent storage (with the limitations already mentioned). Use of the long term persistence scheme for JavaBeans is preferable over serialization for persistent storage. For other persistence mechanisms that can not be modified directly to support the proposed approach for cross-references, one solution is to use the serialization mechanism to create a copy of the substructure to be stored, and the store the copy using the selected persistence mechanism.

These proposed mechanisms can also be used to store the runtime state of the application, but only by storing a full copy of the current application structure. Ready-made support for storing only the runtime state either as a difference from the original state or by storing only the properties marked as “runtime persistent” is left for future extensions of the framework, or can be developed by tool component developers. Tool component developers should ensure that the two default mechanisms are supported for persistent entities that are defined as part of their tool components.

7.4.6 Initialisation and Coordination

JavaBeans anticipates the situation that objects may want to behave differently during design and runtime. The BeanContext contains a designTime property which allows objects contained in the context to discover what the current context of use is. The mechanism defined in the platform independent description allows this status to be part of the application status which can be monitored by any object in the context hierarchy. In a Java implementation, the full application status mechanism should be implemented, but in addition, the designTime property can be supported by mapping from the richer set of statuses supported in the application status.

7.4.7 User Level

The mapping of the user level mechanism to JavaBeans is straight forward as it uses the context service mechanism, and is an application of the change monitoring pattern which maps directly to the JavaBeans event mechanism.

7.4.8 Domain Objects and User-Defined Types and Properties

Java and JavaBeans do not contain any specific mechanism which can be reused for supporting the design of the platform independent model. There is some similarity between the BeanInfo of JavaBeans and the Domain Entity Descriptor, but not enough to merge the two concepts. Instead, the Domain Entity Descriptor can be regarded as an "additional BeanInfo" for domain object types used to support for the user-defined type mechanism.

7.4.9 User-Defined Behaviour

The Java reflection mechanisms can be used by to find the methods and parameters of classes at runtime. It also includes a dynamic invocation mechanism, which allows methods discovered at runtime to be called and method parameters to be provided. In addition, JavaBeans provides facilities for finding properties and events in a similar manner. These mechanisms can be used in the implementation of an end user programming language to interact with the domain objects of the application.

7.5 Editing Framework: Platform Independent Description

The Editing Framework defines the mechanisms and responsibilities for editors that ensure that a set of editor components can be configured to form an editing environment. The goal of the framework is to define the rules necessary to allow smooth integration and collaboration between the components and the environment, and at the same time not introduce too many restrictions that limit the design possibilities. The Editing Framework is based on the Foundation Framework, and thus the mechanisms and services described there should be kept in mind when reading this description. Section 7.5.10 gives a summary of the input used and the rationale of the design for the framework.

An overview of the architecture of the Editing Framework, with the Editing Environment and its main components, is presented in the UML class diagram in Figure 50. The Editor Manager service is a central part of the framework that enforces rules and has a key role in managing the lifecycle of editors. The Action Manager allows editors to interact with the editing actions predefined by the environment, while the Undo Manager handles a stack of undoable commands. The primary extensibility of the Editing Framework is by editor components realising the Editor interface. A secondary extensibility is by components defining policies for how editors are opened in views, and these components must realise the View Policy interface. This is described further in the subsection on view management.

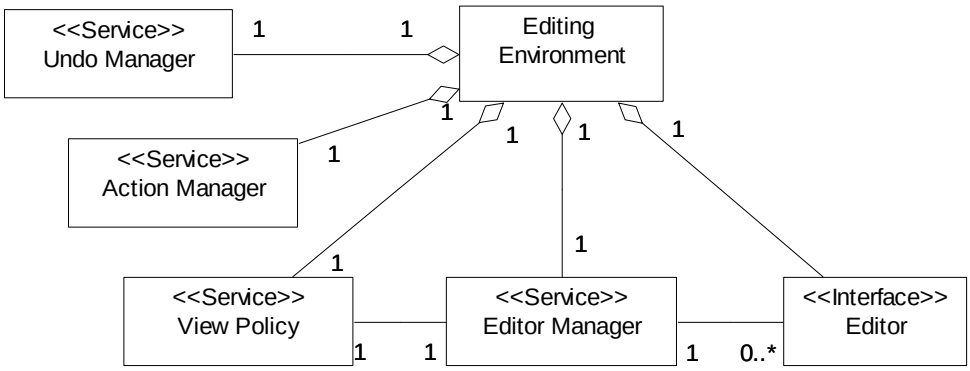

Figure 50 - Overview of architecture of the Editing Framework

The description of the mechanisms and policies defined by the component framework are organised under the following subtopics:

- *Editors and domain objects*. Describes how editors are associated with domain objects, and that categories can be defined to allow different editors associated with an object type.

- *View management*. The framework allows different policies to be defined for how the environment manages the views that editors are contained in.

- *Hierarchical nesting of editors*. Editors are organised in different hierarchical structures that deal with visual containment, services and opening context.

- *Lifecycle of editors*. Deals with when and how editors are created, opened and closed, and which responsibility editors have in these phases.

- *Initial views and editors of an environment*. What is opened when an environment is opened, and how can the editors used in the previous editing session for the application be reopened.

- *Creating and structuring instances and types*. Describes how editors deal with domain instances and user defined types and instances.

- *Collaboration in handling of common actions*. The Editing Framework defines some common actions that most editors will support, and defines how the editors should collaborate in enabling and handling of these actions.

- *Clipboard and drag & drop*. Defines how the editors can handle clipboard operations including export and import to file, and also how to deal with drag & drop operations.

- *Undo handling*. Describes how editors should support handling of undo and redo commands.

7.5.1 Editors and Domain Objects

Each editor in the Editing Framework is opened on a domain entity instance. The editor may be specially made for instances exactly of this class, or may be of a more general nature that can be used with different domain entity types. Some editors may also display substructure of the object within the editor. The editor should give an updated presentation of the domain object(s), and as shown in Figure 51, it uses the mechanisms of the Foundation Framework to register itself to keep track of property changes to the entity, and also react to context changes for the entity (e.g., due to an object being moved or deleted).

The Editor interface defines the method openEditor() that is used to set which object that is currently edited. The object to edit may change during the lifetime of the editor. When the object is set the editor should first cancel registration for notifications from the previous object (if any), and then register to receive notifications from the new object. This also applies to substructures of the object if presented by the editor.

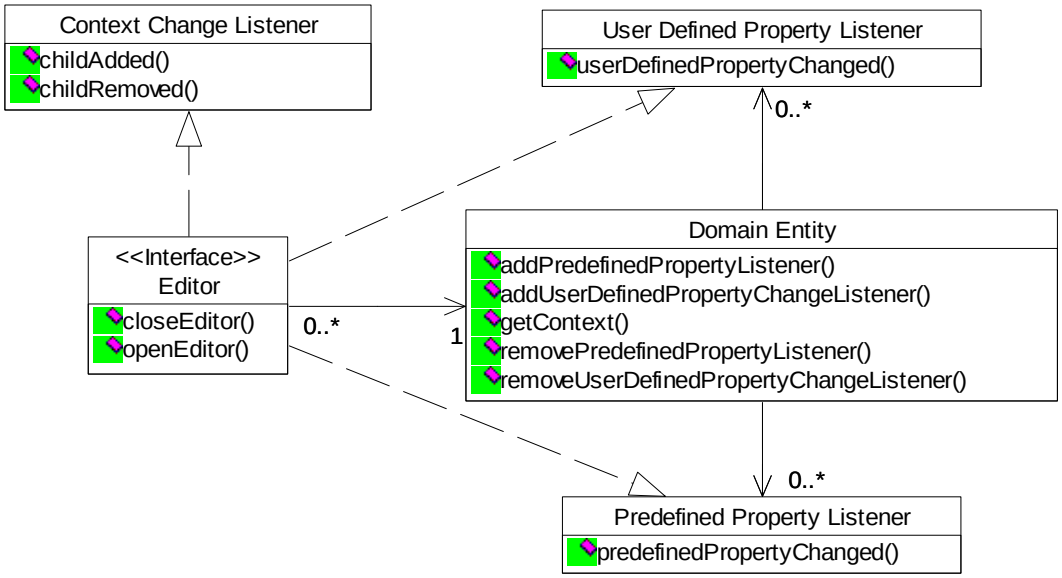

Figure 51 - Editor and update mechanisms of domain objects

The editor to use for editing an entity is selected by the Editor Manager based on a configuration of Editor Association objects that tell which editor type is suitable for an entity type (see Figure 52). In some cases there may be multiple editors for an object type, where the editor preferred to use depends on the situation. A visual container object may, e.g., have an associated visual editor, a tree editor that displays the structure of its content and a detail editor to edit its properties. To support this feature, the Editor Association contains a string that denotes the category the association is for, and when

an editor is opened the preferred category of editor is specified. One of the editors for an object type is marked as default, and this editor will be used if there is no editor of the specified category.

In some cases multiple editors of the same type should display different aspects of the same information, and thus need different configurations. In addition to realising the Editor interface, an editor component usually has an associated class that implement the Editor Config interface. An editor can use configuration info stored in the Editor Config to specialise itself for the situation it is used in. As shown in Figure 52, the Editor Association keeps a reference to the configuration to use for an editor when editing a specific object type and for a specified category. When it is opened, the Editor is notified about which configuration to use for editing an object.

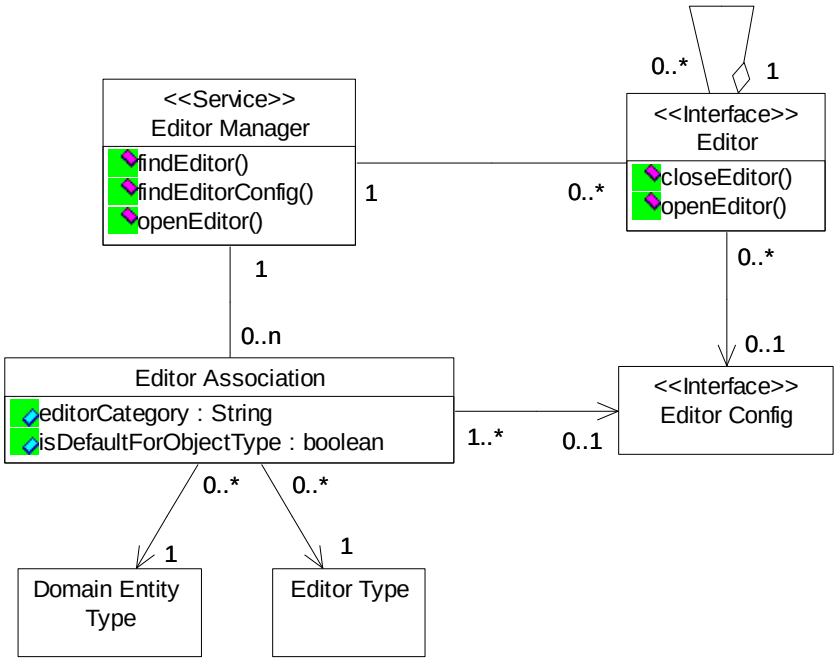

Figure 52 – Use of Editor Association to determine editor type and its configuration

7.5.2 View Management

The view management mechanism takes care of assigning a suitable view to an editor when it is opened. The policy on when to create new views, when to reuse already open views and how to organise the views in windows varies in exiting professional and end-user programming environments. Some examples of policies where the main organising view is a tree view could be:

- use the same view for all editors, and always have only one editor open (in addition to tree view);
- open each editor in a separate window;
- open each editor in a separate view, where the views of editors of the same category are grouped in a window with a tab for each view.

To support different policies for view management, the developer using the Editing Framework can create a View Policy component, or select among one of the predefined policies. Further configuration can be done using the categories assigned in the Editor Association to guide the decisions of the View Policy.

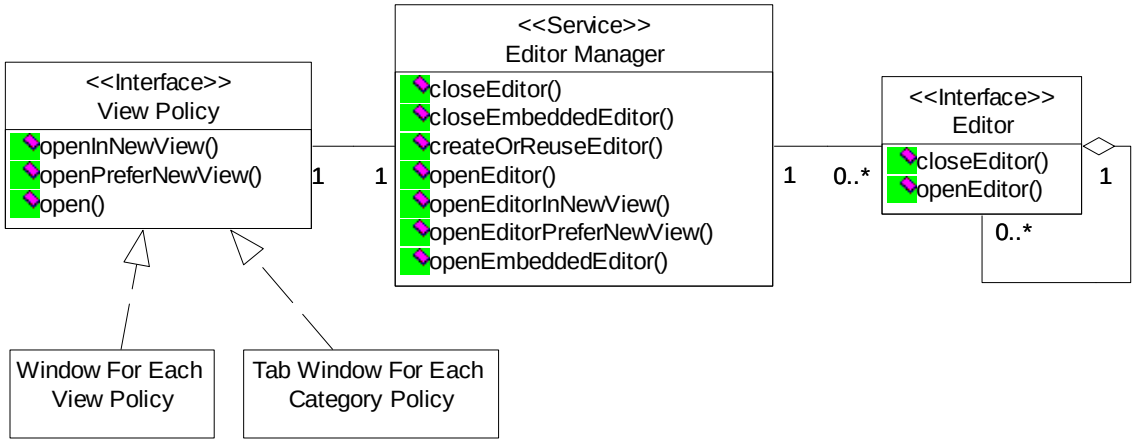

Figure 53 – Editor manager and view policy

The requester that attempts to open an editor can have preferences on its own for where the new editor should be opened. Usually the requestor is another editor. Some editors can embed other editors – this is described in more detail in the next subsection. Embedding editors share part of their own view with the embedded editor and thus the view policy is not involved in this case. When not embedding the new editor, the Editor Manager is requested to open the editor. Figure 53 shows the View Policy interface and the methods used to open and close editors in the Editor Manager and Editor. The open requests can specify view preferences as:

- *Follow policy*. The default approach of the policy will be followed.
- *Prefer new view*. A new view is preferred, but the policy can override this.
- *Force new view*. A new view is always assigned for the editor. As this overrides the policy it can break the user interface consistency of the environment, and should be used with care.

7.5.3 Hierarchical Nesting of Editors

In an editing environment there will often be many different editors open on an application. An editor will in some cases depend on other editors to edit part of the content of the object they are editing, and thus will initiate opening of other editors. The containing editor can, when suitable, embed other editors inside its own view, or it may request that the new editor be opened in a view selected by the current view policy (as described in the subsection on view management).

The editor framework includes each editor in three different hierarchical structures, which are shown in Figure 54. These structures are:

- *The visual hierarchy* handled by the view management, where an editor is displayed in a separate view or embedded in another editor.

- *The opening context hierarchy* which organises how the editors are opened in the context of other editors.

- *The hierarchical context* which organise service access.

The visual hierarchy should build on the mechanisms provided by the target platform, and are not specified in detail here. In general, though, the editor should provide some way to get access to its corresponding visual component. Similarly, each view provided by the view management mechanism should give access to the corresponding visual container.

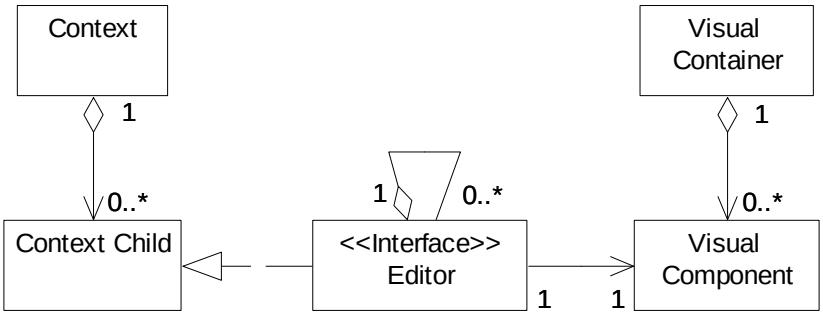

Figure 54 – Hierarchical nesting of editors

When an editor wants to embed another editor, it requests the Editor Manager to create a suitable editor of a specified category for a domain entity. It also provides the visual container that the visual component of the new editor will be embedded in. The containing editor is responsible for notifying the Editor Manager when an embedded editor is closed, both if the reason for closing the editor is internal to the containing editor and if it is because the containing editor itself is closed.

The Editor Manager is also used when an editor should be opened in a view. Editors opened in views are managed directly by the view system, and thus the editor opening these does not have any further responsibility for notifications etc.

The opening context hierarchy of an editor is the sequence of “ancestor” editors that should already be open to present the editor in the right setting for the application developer. When an editor is embedded in another, the embedding editor is also the opening context. If the editor is presented in a view it may still have an opening context in another view. The opening context of an editor is passed as a parameter when openEditor() is called.

The primary use of the opening context hierarchy is to enable creation of bookmarks that can be used to reopen an editor on a specific object within the right context. Bookmarks can be used directly in the environment similar to what is done in web browsers, but can also used by undo operations to make sure the editor where the undoable operation was performed is open. A bookmark is kept as an ordered list of bookmark nodes, with the node for the root editor of the context as the first node. The

editors in the opening context collaborate on creating bookmarks and on opening an existing bookmark.

The editors are also organised in a hierarchical context for service management, using the mechanism defined by the Foundation Framework. The context of the editors gives access to any services provided by editors and views higher up in the hierarchy, and also to services provided by the editing environment. The editors are added and removed from their context by the Editor Manager, and should not themselves do any changes to the context hierarchy. As long as the context of the editor has a non-null value, the editor can access services from the context.

The service context of an editor may be an embedding editor, the view the editor is displayed in, the window containing the view, or the editing environment. It is up to the view policy to decide if a view or window will be a context or if the editor is connected directly under the editing environment. An embedding editor should set itself or its own context as the context of embedded editors. This arrangement allows embedding editors, views, windows and the editing environment to provide services to the editors. Also a context closer to the editor may override the services provided higher up in the hierarchy. An example of this could be that the window the editor is contained in provides a palette service with the palette displayed in the window. This would override a palette service provided by the editing environment, where the palette would be common to all windows.

7.5.4 Lifecycle for Editors

Instances of editors are only created by the Editor Manager. The request to create an editor instance usually comes from another editor that is either embedding the new editor or is a structure editor (e.g., tree editor).

After an editor is created, the openEditor() method will be called, telling the editor which object to edit and optionally providing a configuration for the editor that is suitable when editing this object. Before the openEditor() method of an editor is called, the editor will be added to the hierarchical context, and the implementation of the openEditor() method can safely request services from the context. The editor may or may not be in a visual container when openEditor() is called, and thus should not rely on this in the implementation. The openEditor() method may later be called again to set a new object and configuration for the editor, and the same rules apply in this situation. Figure 55 shows a UML sequence diagram with an example of the message flow resulting from opening an editor.

The closing of an editor may originate from different events:

- the view (and window) containing the editor is closed;
- another editor should be opened in the view replacing the current editor;
- the embedding editor decides to close an embedded editor, e.g., because it needs the space for another editor;
- the embedding editor is closed, so all embedded editors are closed as well.

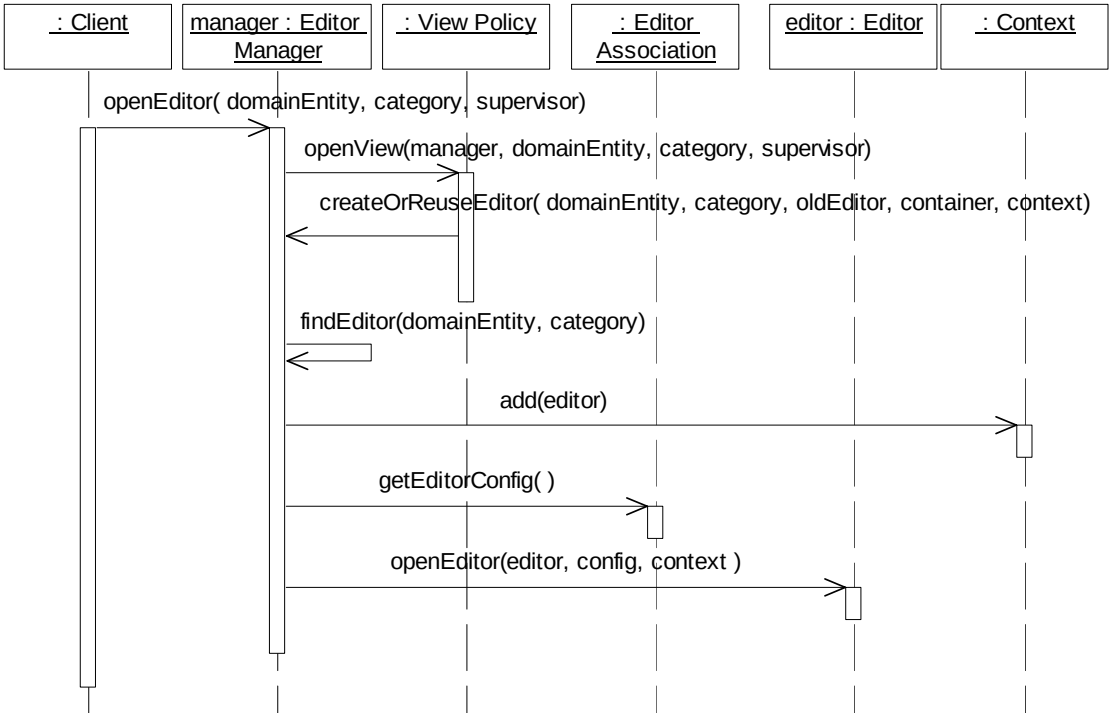

Figure 55 – Example of sequence for opening editor

In all cases, the Editor Manager must be notified when an editor should be closed. It is the responsibility of an embedding editor to notify the Editor Manager when an embedded editor is closed, and this applies recursively. The view policy is responsible for making sure that the views and windows it uses will notify the Editor Manager when they are closed. When a view is being reused for a new editor, the Editor Manager will take care of closing the previous editor contained in the view first.

When an editor is closed, it is first removed from the visual container by the Editor Manager before the closeEditor() method of the editor is called. Thus the editor is not part of its visual container when closeEditor() is performed. The editor will be removed from the hierarchical context after the closeEditor() method has been called, and thus the context is still valid in closeEditor().

7.5.5 Initial Views and Editors of an Environment

When a new application is opened in the editing environment, an editor for the root object of the application should automatically be opened by the environment. The category of editor to use for this purpose is determined by the default among the Editor Associations for the root object type. The view policy of the environment will set up the window to use for the editor.

Existing environments vary in their behaviour for what is opened when reopening an application that has previously been edited. Some environments will just open a root editor, while other reopens the same views and editors that were open when the application was last edited. A view policy could store information about which views and editors are open, and restore this on request. The bookmarks used for keeping track of the opening context of editors can be used to implement this.

7.5.6 Creating and Structuring Instances and Types

Some of the editors created with the framework will allow the application developer to create new instances and types. There are two types of objects that can be created: instances of user-defined types, and new user-defined types. Instances of domain object are for uniformity required to always have a corresponding user defined type.

The domain frameworks developed using the frameworks should be extensible, and the main kind of extension that is made is new object types. The set of available types for creating instances thus have to be discovered when the editing environment is running. As described in the Foundation Framework, the available types in the system can be found through the component registry.

The set of suitable object types from which new instances can be created, are usually presented as part of a palette or in a popup-menu for the editor. When an instance of a user-defined type should be created, all user-defined types supporting a specified set of interfaces are potentially usable. The domain may define additional rules that should be applied, e.g., to only include types that are defined in a specific subpart of the application, and an editor may allow the environment developer to customise this using the Editor Config. An editor may also include more advanced facilities, for example, providing a wizard to create and edit initial values of an instance, or even create a whole structure of instances.

When creating new user-defined types, the application developer selects among a set of predefined domain base types for user-defined types, or starts from a copy of an existing user defined type. As described in the Foundation Framework, the instances of the new type are defined as instances of the selected base type.

7.5.7 Collaborating in Handling of Common Actions

The editing environment defines a set of common actions that are either supported directly by the environment, or supported by most editors. These actions are available in a menu bar and/or toolbar that is set up by the environment or the view depending on the view policy.

The actions are organised in two groups: file related actions and editing actions. The file related actions are usually presented in a file menu. These are handled by the editing environment itself, and triggers the persistent storage solutions where required. The editors of the environment are not directly involved in these actions. A typical version of the file menu is displayed in Figure 56.

The editing actions are usually found in the edit menu as presented in Figure 57. These actions are of more direct concern to the editors. The undo and redo actions are handled by the environment, but based on the undoable edits generated by the editors, as explained in the undo subsection (7.5.9). The edit menu also contains the standard clipboard actions with additions for copy and paste to file as described in the clipboard and drag & drop subsection.

The last two edit actions are select all and find. The exact behaviour of these operations depends on what is natural to support for the editor. Select all can, e.g., select all text in a text editor that is part of an editor, or select all nodes in a tree editor. The find action

triggers a search in the current editor, e.g., to find an object based on its ID or to find text content.

Figure 56 - File menu

Figure 57 - Edit menu

As the edit actions are shared among all editors, a system is needed for determining when an editor is the target of the actions. In the Editing Framework this is solved by having at most one active editor at all times. An editor is activated whenever its component or a component contained in it receives the keyboard focus. With nested editors, the editor with the component that is the closest ancestor of the focused component is activated. It is the responsibility of the Editor Manager to activate and deactivate editors. This is similar to the approach in OpenDoc.

When an editor is activated, it should register itself with the Action Manager for receiving edit action notifications, using the Edit Action Listener interface as shown in Figure 58. As long as it is active, the editor is also responsible for enabling and disabling the edit actions. The Action Manager is a service, and can be located through the editor's service context. Undo and redo are enabled and handled by the environment.

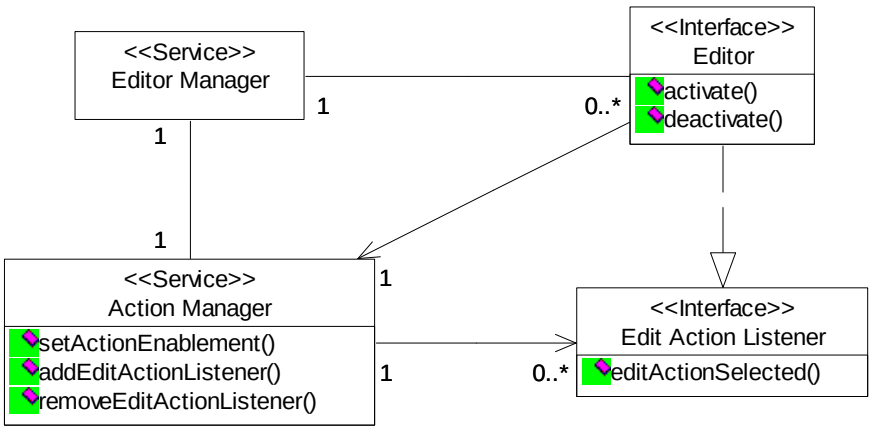

Figure 58 – Management of editing actions and editor activation

7.5.8 Clipboard and Drag & Drop

It is preferable, but not required, that editors support clipboard and drag & drop operations where this is natural in the user interface of the editor. The support for these operations should be based on the standard support provided by the component technology platform.

Standard menus and/or toolbar items for the standard clipboard operations are also provided by the editing environment, and these can be accessed and managed by the editor as described in the previous section. The editor may also provide its own user interface to clipboard operations. This is usually done by including the operations in a popup menu.

During the data transfer of clipboard and drag & drop operations, both regular and weak references have to be handled correctly. This problem is the same as during persistent storage handling, and can be handled using the same mechanisms.

The Editing Framework also provides support for copying and pasting content to file. It is up to the editor to decide if these operations should be available for its content. Supporting these operations is simple, but care should be taken to enable these operations only where really useful to avoid confusing the application developer with too many different kinds of stored object files.

7.5.9 Undo Handling

The undo mechanism of the Editing Framework is inspired by the undo support of Java's Swing library. The editing environment uses an Undo Manager component to keep track of the current undo/redo stack. When an editor performs an edit that is undoable, it must generate an undoable edit event. The Editor Manager adds and removes the Undo Manager as a listener of undoable edit events with editors when they are opened and closed. It is up to the editor to decide which edits should be undoable, but as a general rule, all edits that causes changes that is difficult or time consuming to reconstruct for the user should be undoable (although in some domains, such as accounting and electronic patient journals, undo is not allowed). The main actors involved in the undo handling are shown in Figure 59.

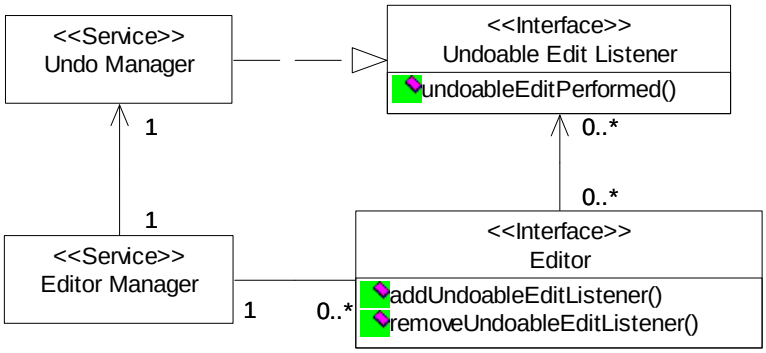

Figure 59 – Main actors involved in undo handling

An undo operation has two things to restore: changes to the structure of domain objects, and changes within the editors (e.g., change of editor, selection etc). The changes to the

domain objects have to be restored, while changes within an editor are preferable to restore so that the user can look at the exact situation before undo was chosen. The bookmarks used to store an editor opening context can be used to reopen the right editors. Each bookmark node also allows the editor to store editor specific information that can allow it to restore selections etc.

7.5.10 Rationale for the Design

While the Foundation Framework to a large extent refines and reuses mechanisms from the cases studies, the Editing Framework has not to the same extent been based on existing mechanisms. The Editing Framework was primarily designed to address the issues of how a set of editors, some of which may address the same areas of concern, can be integrated into a single environment. This was identified in Chapter 6 as a central challenge to handle. Still, experience from the case studies was used as input to the design, and also some inspiration has been drawn from OpenDoc. From the tasks descriptions in Chapter 5, the editing framework has primarily related to the *Edit Application* use-case (and its detailed tasks) of the application developer.

The Foundation Framework already contains part of the support required for the *Edit Application* use-case. The primary addition which the Editing Framework defines for this use-case is how the editors relate to the domain objects, and how the editors are organised and collaborates to form a consistent environment. This level of detail is not specified in the use-cases, but the mechanisms contribute to how usable the environment is for the application developer in performing the tasks.

Some of the issues covered by the Editing Framework, such as support for clipboard and undo operations, are generic issues common to most applications with an editing element, and thus these issues will be addressed also in other frameworks supporting editing. These issues were still included here, both because they are important aspects of the framework, and because the relation to the Foundation Framework and other parts of the Editing Framework should be considered.

7.6 Editing Framework: Mapping to Java/JavaBeans

This section describes how the mechanisms from the platform independent description of the Editing Framework can be mapped to Java and JavaBeans. Descriptions are only included for the mechanisms where platform specific issues were found. In addition to the described mechanisms, it should also be noted that the editors edit structures of domain objects, which are also JavaBeans. Thus the editing environment may utilise the mechanisms defined in JavaBeans. The BeanInfo, e.g., defines icons for the JavaBeans in two different sizes, and these may be used by the editors when presenting domain objects.

7.6.1 Editors and Domain Objects

One thing that has been considered when designing the editor system is if the editors should implement the Customizer interface defined in Java. While this may be reasonable for some of the editors built using this framework, generally the editors are depending on services and features of the Foundation and Editing Frameworks that are

not available if an editor is attempted reused outside the frameworks. To avoid signalling a potential reuse when it is unlikely to be successful, the editors of the framework should not implement the Customizer interface unless they are usefully reusable outside the frameworks.

7.6.2 Hierarchical Nesting of Editors

In the Java mapping, the visual hierarchy is built on Java's visual components as defined by the Component and Container classes in Java's Abstract Window Toolkit (AWT). An editor is not itself required to be a subclass of the java.awt.Component, but instead the Editor interface has a getComponent() method. Associated with each view from the view management mechanism is a java.awt.Container that will be the container of the visual part of the editor.

As defined by the Foundation Framework mapping, the BeanContext of JavaBeans is used for realisation of the service context mechanism.

7.6.3 Clipboard and Drag & Drop

Clipboard and drag & drop support should in Java be built using the standard Java support in this area (found in the java.awt.datatransfer and java.awt.dnd packages).

7.6.4 Undo Handling

As the design of the undo mechanism is based on the undo support of Java's Swing library, the mapping to Java in this area is straight forward.

7.7 Summary

This chapter has proposed an overall architecture for building document-based end-user programming environments, including an outline of a two component frameworks. The overall architecture is based on a non-strict layer model. The layer model defines how the different frameworks and components relate to, and build on each other, and how runtime and edit time components are built on top of these.

The Foundation Framework specifies additional services and rules beyond the standard component platforms, and provides a foundation on which the Editing Framework and domain oriented frameworks for document-based end-user programming environments can be built. In the Foundation Framework, patterns for change monitoring and repositories are described and repeatedly used to define its key mechanisms. The framework defines how domain objects and user defined types are handled, and how user defined behaviour can be attached and triggered. A hierarchical context mechanism forms the navigable structure for the domain objects, and defines how services are managed and discovered. An object repository organises the domain instances, and allows retrieval by name as well as the set of interfaces supported by the objects, while a component repository keeps track of the components available in the system. Mechanisms suitable for data transfer and document-based storage of the object

structures were discussed, including how the containment and cross-references between domain objects can be handled.

The Editing Framework defines rules and components that support the coexistence and collaboration between a set of editors in an editing environment. This framework defines how editors relate to domain objects, and how they are included in different hierarchical structures. The assignment of editors to views is subject to the selected view policy of the editing environment. Lifecycle management for editors are described, including how initial views and editors of an editing environment are set up. The framework also describes how editors are involved in creating instances and types, how editing actions common to all editors are handled, and how undo support, clipboard and drag&drop are handled.

For both the Foundation Framework and the Editing Framework, a platform independent description was given, before a mapping was done to Java/JavaBeans as an example of a target platform.

Chapter 8 Tool Support

This chapter describes how a set of tools can be used to support the development of domain-oriented application composition environments based on the overall architecture and component frameworks proposed in the previous chapter. It thus addresses the research question: *What kind of tools will assist developers in building such environments?* The primary focus is on what kind of tool support that is useful for the domain environment developer when creating environments based on these frameworks. The use-cases of the domain environment developer described in Section 5.3 are used to consider which tasks the tools should support. The focus has further been on defining tools which are feasible to create and which raise the abstraction level of the development work, either by reducing the distance from design model to a running implementation or by transforming manual coding tasks into configuration tasks. This chapter also includes brief discussions of tools for the domain component developer and the tool component developer. The last section of this chapter gives more details about which parts of the proposed frameworks and tools that have been prototyped.

8.1 Tool Support for the Domain Environment Developer

This section describes tool support for tasks performed by the domain environment developer. The description here is organised following the domain environment developer's main use-cases, as described in Section 5.3:

- *create domain framework;*
- *create domain composition environment;*
- *create domain runtime environment;*
- *create extensibility support.*

8.1.1 Tool Support for Creating Domain Frameworks

The main activities involved in creating a domain framework were presented in Section 5.3.2 and were summarised in Figure 28. Most of the activities described are analysis and design activities. As suggested in this description, a suitable analysis and design method (e.g., OORAM or RUP (Rational Unified Process)) should be used for this work, and a modelling tool supporting a suitable notation (e.g., UML) should be used to create models of the domain framework. The design work is also guided by the

restrictions and possibilities defined by the mechanisms and rules of the Foundation Framework.

The model of the domain framework that is created through the analysis and design activities is a blueprint for the implementation of the domain framework. Although the implementation activity can be based on a manual transformation of the model, there are good reasons for automating parts of the transformation. A direct connection between the model and the implementation helps to ensure that the consistency between the model and implementation is maintained. Also, it directly pays off with respect to the effort that is put into creating the model in the first place. A disadvantage is that an automated transformation does not give the same degree of freedom in the mapping from model to implementation.

The domain model is an abstraction of the domain framework, and thus only parts of the implementation of the domain classes can be precisely derived from the model. UML models can be made more precise through the use of stereotypes for classes, methods, and attributes, which define more of the semantics for the stereotyped model elements. Table 11, Table 12 and Table 13 define stereotypes for classes, attributes and associations which are used to relate model elements to concepts of the Foundation Framework. These stereotypes were derived by analysing the Foundation Framework to find higher level modelling elements with semantics defined by the mechanisms of the framework.

Table 11 - Class stereotypes based on the Foundation Framework

Class stereotype	*Description*
<<Domain Entity>>	A persistent entity within the domain from which the application developer creates instances, and which is used as the basis for user defined types. Follows the behaviour defined for Domain Entity and Context Child in the in the Foundation Framework.
<<Domain Container>>	A Domain Entity that is also a container of other domain entities, and supports the behaviour of a Context as defined by the Foundation Framework.
<<Service>>	A tool without persistent content of its own that contains the necessary support for being a service accessible through a context as defined by Foundation Framework.

Table 12 - Attribute stereotypes based on the Foundation Framework

Attribute stereotype	*Description*
<<Shared>>	The attribute is a shared property as defined by the Foundation Framework

<<Transient>>	The attribute is not part of the persistent state during edit or runtime.
<<RT_State>>	The attribute is part of the persistent runtime state of the application (e.g., game save state)

Table 13 - Association stereotypes based on the Foundation Framework

Association stereotype	*Description*
<<Service provided>>	Used from a Domain Container to a Service to specify that the container provides the service as defined in the Foundation Framework.
<<Service use>>	Used from a Domain Entity to a service to specify that the entity will acquire and use the service as defined by the Foundation Framework.

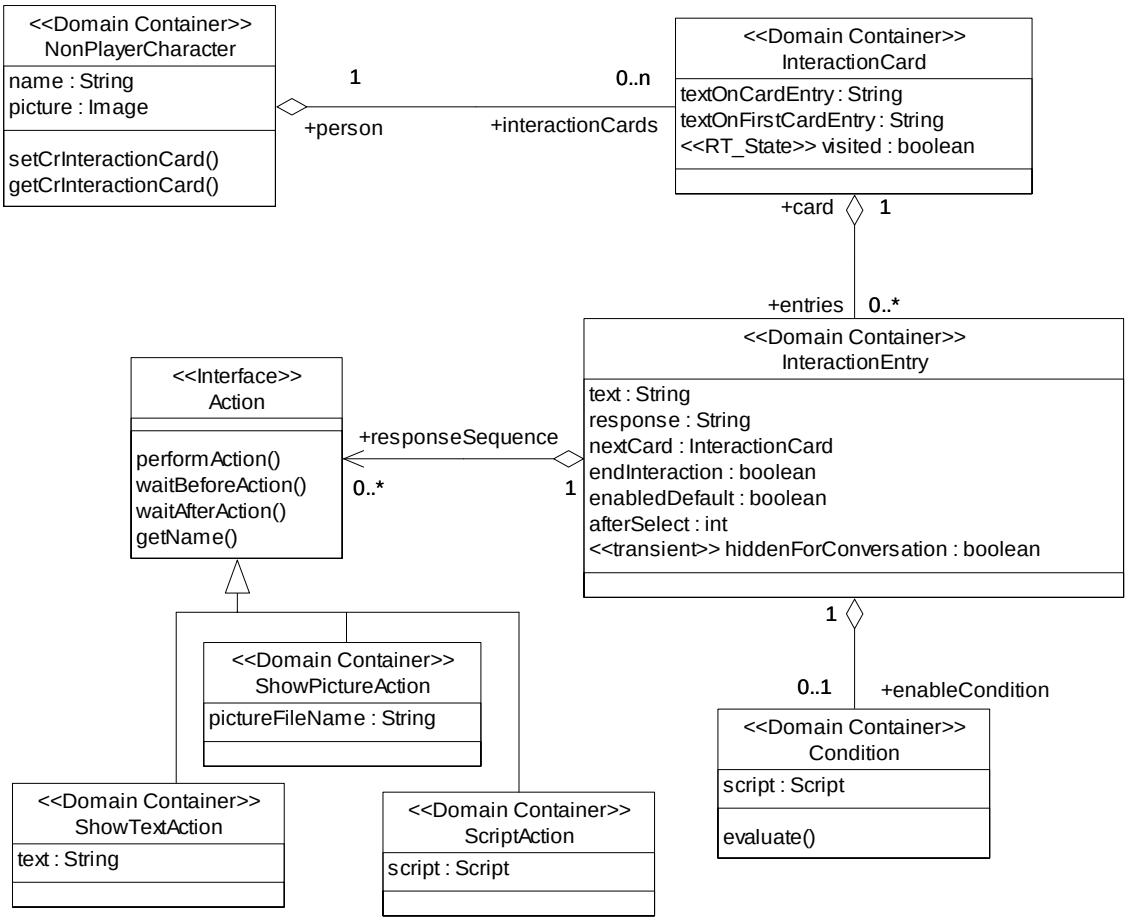

Figure 60 - Stereotypes applied to role playing game example

Figure 60 shows an example of use of some of the stereotypes applied to the dialog part of the role playing game example.

State-of-the-art modelling tools usually contain some support for transforming the models to an implementation, or have special versions or add-ons to the tools for this purpose. This support has usually ranged from scripting languages allowing the developers to create their own mapping to implementation, to predefined mappings for specific languages and environments (e.g., C++, Java, Enterprise JavaBeans). Recently, the leading modelling tool vendors have also added further support for the Model Driven Architecture (MDA) approach (see Section 9.2.1 for further relation to MDA).

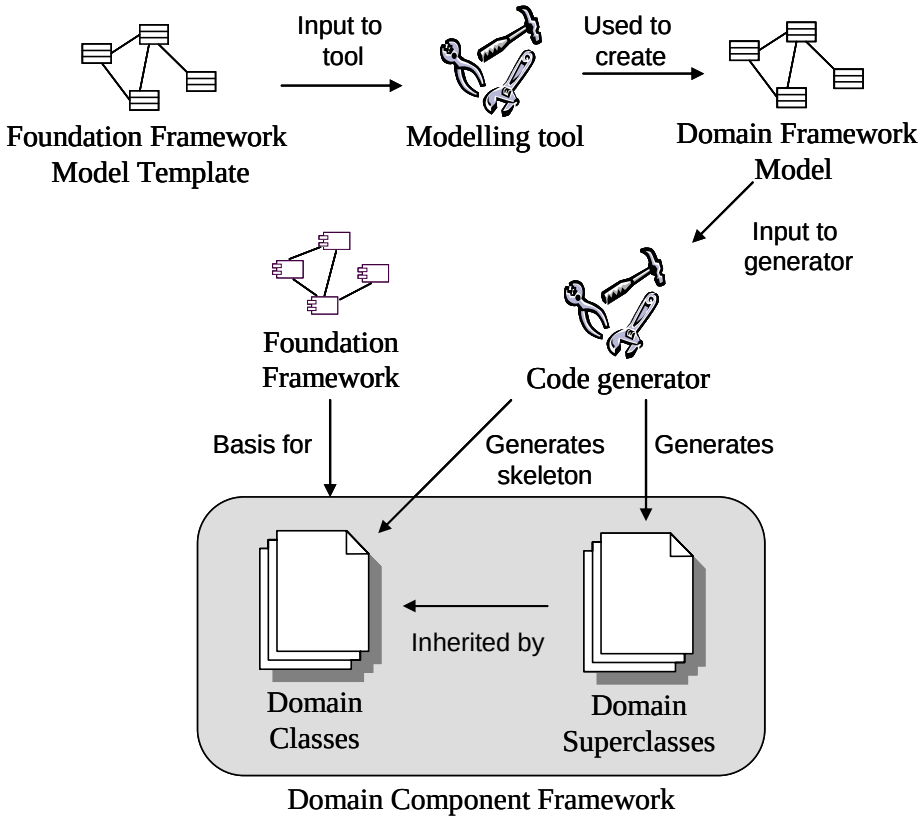

Figure 61 - Artifacts and tools for model driven domain framework development

Figure 61 shows an example of how tools and artifacts can be related in a model driven approach to creating a domain framework. Here, a model template for the Foundation Framework is used as input to the modelling tool, and defines stereotypes and class libraries which are used when creating the domain framework model. A Code Generator tool uses the domain framework model as input, e.g., in the form of an XMI format model file (the XML-based standard interchange format for models and other metadata). From the model, the generator in this example creates superclasses for the domain types which include the predefined behaviour that can be derived from the model. Also, it can generate skeleton classes inheriting from these superclasses, which the developer uses as a starting point for the hand-coded parts of the implementation. This approach allows the generated behaviour to be updated when the model changes, without destroying the hand-coded implementation. In addition to the tools covered in the figure, the developer will of course also use a programming language tool, probably from within an integrated development environment. The connection between model

and implementation is further improved if the code generation is included in the make process of the selected tool, as this helps to encourage the developer not to make manual changes directly on the generated code.

Possible variations and refinements of the above approach include:

- The Code Generator tool can either be implemented as a script that can be executed from the modelling tool, as a customisable mapping tool, or as a separate program. In any case, there can be generators for different target platforms.

- Some modelling tools also support reverse transformations from implementation classes to models, and, e.g., allow the classes in the model to be updated with attributes and methods added in the implementation. Although this may be useful in some situations, such reverse transformations may introduce unwanted implementation details to the model, and may have problems discovering higher-level concepts (such as Stereotypes) from the implementation. Also, useful diagrams can usually not be automatically generated by such approaches.

Table 14 describes a mapping which a generator can use to generate code for a Java/JavaBeans-based implementation of the Foundation Framework, based on the domain framework model. The table is organised in sections related to classes, attributes and relations, and the most general concept is presented first in each section.

Table 14 - Mapping from model concepts to Java-based implementation

UML concept / Stereotype	*Implementation based on a Java/JavaBeans-based version of the Foundation Framework*
Class	Maps to a Java class. Generates a superclass for the specified class, and optionally a skeleton for the class itself set up with the correct inheritance and with method skeletons
Interface	Maps to a Java interface
<<Domain Entity>>	Maps as class, but with the following additions. Generates necessary meta-information, including domain entity descriptors and their contained property descriptors and support for creating user defined types. The generated superclass supports the context mechanism by implementing the BeanContextChild interface. Change listener supported for properties are also generated, as well as support for user defined properties including delegation. Persistence support can be generated based on definition of associated attributes and relations.
<<Domain Container>>	As Domain Entity, but extends the generated support in the superclass to also include handling of the BeanContextServices interface. Handling of provided services and contained objects are added as required service

	provided relations and aggregation associations.
<<Service>>	Maps as Java class. In addition superclass implements empty Service marker interface.
Attribute	Mapped to property in the JavaBeans model, and generates read-method, and write-method with change notification. By default, persistence design-time persistence support is included in the superclass for Domain Entities, and for each attribute a corresponding property descriptor is added to the domain entity descriptor.
<<Transient>>	Attributes with this stereotype is marked with the Java transient keyword, and is omitted from the persistence support generated in the superclass. This will also be reflected in the corresponding property descriptor.
<<RT_State>>	For attributes with this stereotype, the generated superclass will have support for storing the attribute as part of runtime state, and this will also be reflected in the corresponding property descriptor.
<<Shared>>	For attributes with this stereotype, code is generated to support delegation of property requests to the user defined type, and this will also be reflected in the corresponding property descriptor.
Association	Mapped to a property. Aggregation associations are used to express containment in the domain model, so that implementation mappings can distinguish between containment and cross-reference behaviour for persistence and data transfers and generate the corresponding code. Aggregations also generate suitable container support for the contained element, including context change events when contained elements are added and removed.
<<Service provided>>	Generates code in the Domain Container and an associated BeanContextServiceProvider for giving access and providing notifications about the specified service.
<<Service use>>	Generates code in the Domain Entity for getting hold of a reference to the service through the context, and reacting to change notifications for the service.

8.1.2 Tool Support for Creating Domain Composition Environments

The Editing Framework presented in the previous chapter defines how a set of editors can collaborate in a composition environment. The rules of the framework help to ensure that a set of independently developed editors can be combined in a composition

environment. A composition environment for a specific domain can contain a combination of reusable editors (for example, the kind of editors presented in Chapter 6) and editors made specifically for the domain.

The detailed diagram of the create domain composition environment use-case presented in Figure 29 divides the work into *configure structure editors*, *configure property editors for domain classes*, and *configure points of customisable behaviour*. The domain environment developer can in theory perform these tasks without further support using the Foundation and Editing Frameworks, and a combination of reusable and special made editor components. To assist the developer further, though, a set of tools based on the Editing Framework is proposed. These tools help to change the nature of the tasks from coding to configuration, allowing the developer to create environments with less effort.

The Environment Builder is a tool assisting the developer in creating a domain composition environment. When creating a new composition environment, the domain model to build an environment for is first imported into the tool. In this way, the domain classes and their associations, attributes, etc. are known by the tool. The tool allows the developer to define the set of editor categories to use in a composition environment (see Figure 62). Each domain class can have an associated editor for each defined editor category (see Figure 63), basically providing an editor for setting up the EditorAssociation objects defined in the Editing Framework. These definitions decide which editors are opened, and when they are opened for the domain editing environment that is being developed.

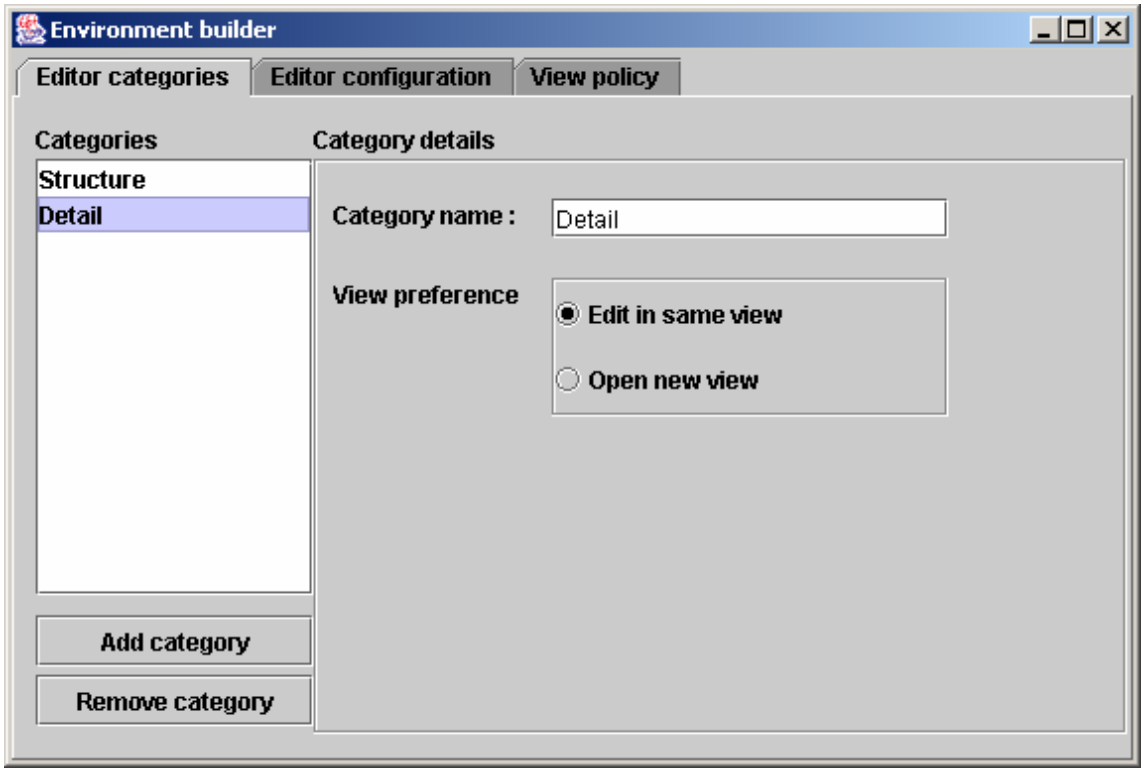

Figure 62 – The Environment Builder user interface for defining editor categories

The Environment Builder supports an extensible set of editors. Each editor may allow configuration for a specific use, and the EditorAssociation can hold an EditorConfig

object for this purpose. The range of configuration can be from fine-tuning and visual preferences in some editors, to more fundamental setup in other editors. To perform this configuration, the Environment Builder allows each editor to have an associated meta-editor. The meta-editor appears in the area shown at the bottom of Figure 63. Currently this is done by requiring that the editors either have a JavaBeans customizer or that the configuration can be performed with a property sheet. In theory, the Editing Framework could have been applied also on this meta-level, with "editor configuration" as the domain. Although it would be possible and would be a nice proof of concept, it is not necessarily the right choice for how to make a tool for the user group: the domain environment developers. Further considerations on if and how to do this has been left for future work.

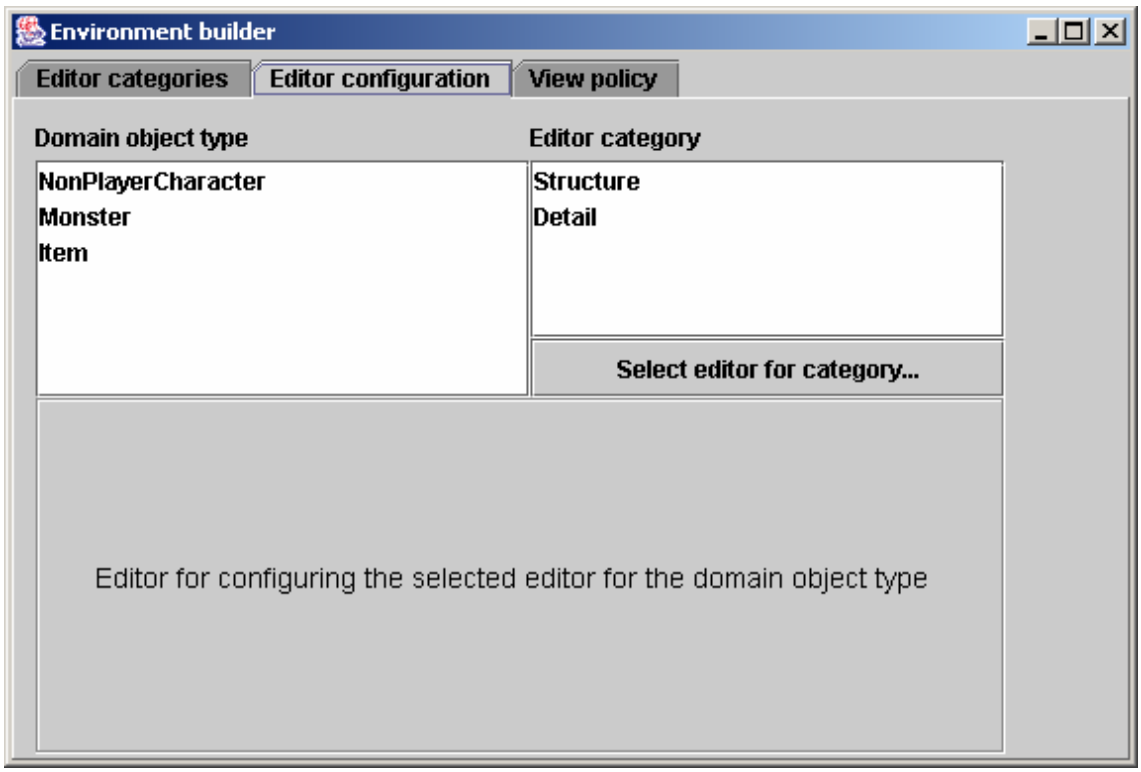

Figure 63 – Selection of editors for domain types in the Environment Builder

An editor for a domain object can be implemented as a class specific for the domain object type. For many of the domain object types, a form-based interface will be suitable. One of the predefined editors of the Environment Builder allows the developer to quickly configure a form-based interface for a domain object type. This generic Form Editor lets the developer pick properties from the domain object type, and thus supports the *configure property editors for domain classes* use-case. Based on the type of each property the developer can chose among a set of available property editors for the type. E.g., for a boolean property a checkbox, radio-button or combo-box property editor can be chosen, and for a string property, a single line input box or a multi-line editor can be used. The Form Editor is itself extensible with new component types that can add to the set of property type that can be edited or add new kinds of editors for existing property types.

The Form Editor, with its placement within the Environment Builder, is displayed in Figure 64. The Form Editor appears within the editor configuration area when the

environment developer has selected to use this editor for a specific editor category of a domain object type. The list at the left of the figure displays the properties and associations defined in the domain model for the selected domain object – in this case the InteractionEntry from the role playing game example. From this list, properties can be copied to the list in the middle, and a suitable default editor for the property will be selected. The field at the right allows further configuration of how to edit a selected property, and the editor preference box in this editor can be used by custom property editor components. The Form Editor also allows definition of groups of properties, with each group being presented in a tab sheet in the resulting editor in the domain environment.

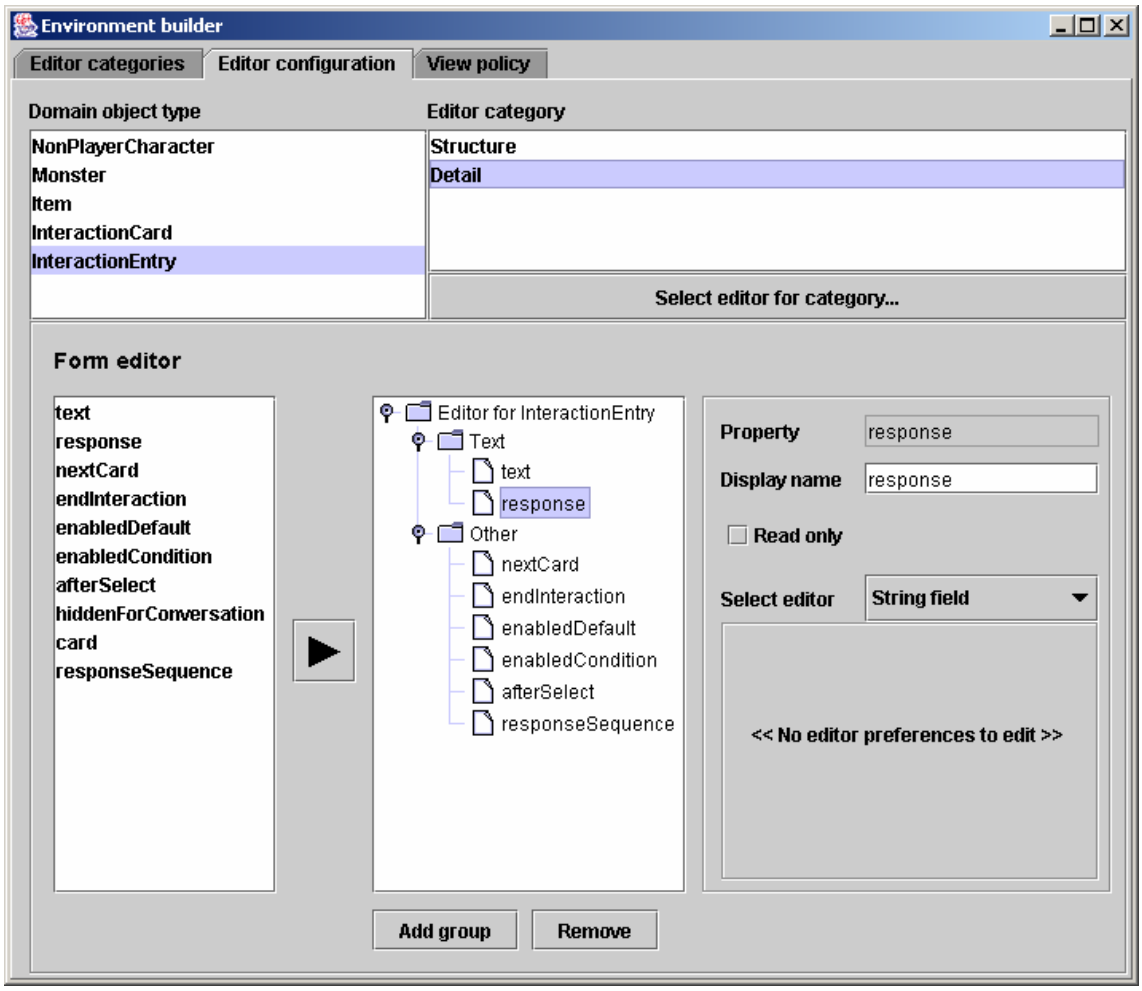

Figure 64 – The Form Editor component displayed within the Environment Builder

Other types of editors that require some configuration are structure editors; e.g., tree editors, user interface editors, and grid-based editors. These editors differ from the property-based editors like the Form Editor in that multiple kinds of objects are typically displayed at the same time in the editors. These editors may need configuration for each domain object type to specify which sub-structures are included in the view, in which sub-structures new instances can be added and of which domain object types, etc. To simplify the configuration, the editors can use techniques like configurable default behaviour that applies when no specific configuration is done for a domain object type, inheritance of configuration from other object types (explicitly defined or implicitly through Java class inheritance), etc.

The structure editors should also define where it is possible to define new user defined types. The mechanisms for allowing definition of user defined types is provided by the Foundation Framework, and the domain framework defines the base types from which the user defined types can be derived. Behaviour can be connected to domain objects by associating it with an event, or by using predefined properties of domain objects that will hold some kind of "triggerable" code; e.g., an attachable precondition or a configuration that will be applied at a certain point during runtime. Configuration of editors for such properties can be included in the same way as the editors for more simple property types. Similarly, the Form Editor can be extended to also present the events associated with a domain object type, and to allow selection and configuration of behaviour editors for these.

The Editing Framework uses a view policy to decide how the different categories of editors are mapped to views. Figure 65 displays how this is integrated in the Environment Builder. A view policy may require some additional configuration, and the figure displays the configuration possible for the "Window for each category" view policy, where each editor category is mapped to one of two predefined views.

The Application Developer uses the Generic Editing Environment tool to execute an editing environment created using the Environment Builder. This tool will create a runtime instance of the Editing Framework, and uses the definitions created with the Environment Builder to set up the editing environment. The Generic Editing Environment provides the standard file and edit menus, including undo/redo, by utilising the Action Manager, clipboard and undo handling functionality provided by the Editing Framework. The application also includes a component manager component that allows the application developer to add and manage components created for the editing environment by third parties. Figure 66 shows a screenshot from an early prototype of the Generic Editing Environment running the example editor created by the Form Editor in Figure 64.

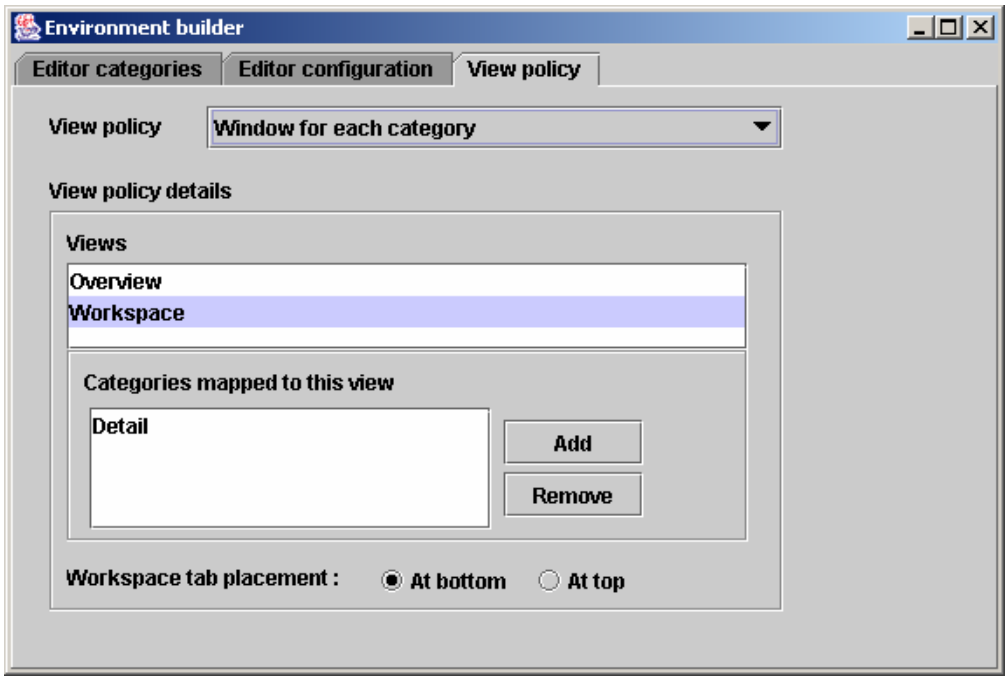

Figure 65 - Configuring views in the Environment Builder

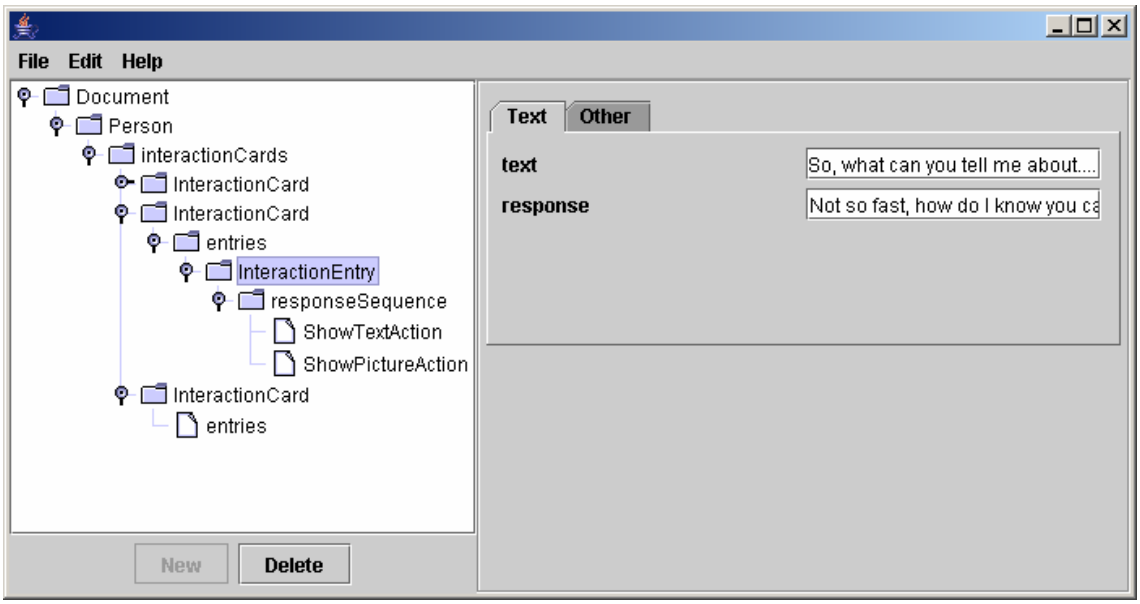

Figure 66 – Prototype of Generic Editing Environment

To summarise, the Environment Builder is used to associate and configure the editors for use with editing domain objects, and to configure the view policies of the editing environment, while the Generic Editing Environment is used by the Application Developer to execute the environment. Figure 67 show how the relations between the different models, components, configurations, frameworks and tools used to create a domain specific application composition environment.

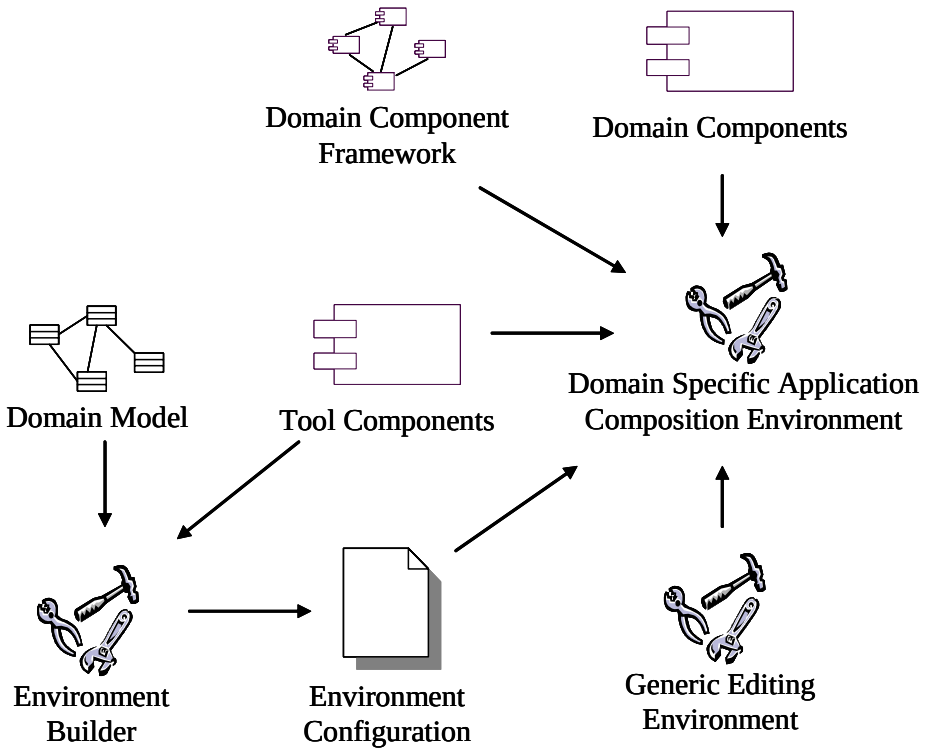

Figure 67 – Relations between tools and their required input

8.1.3 Tool Support for Creating Domain Runtime Environments

To execute an application created by the application developer, a domain runtime environment is needed. As an example, in the role playing game example this would be the game engine and user interface which is used to interpret and play the game story defined be the application developer. A central part of the runtime environment is the domain component framework, for which tool support was described in Section 8.1.1. Through the Foundation Framework, the domain framework and the runtime environment have support for handling runtime issues such as user defined types, hierarchical context management, services and persistence of application runtime state.

The domain composition environment, which is created using the Environment Builder, will include a set of behaviour editors which the environment will support. The runtime environment needs corresponding components for runtime behaviour representation and execution for behaviour created by the application developer. As the execution model of different behaviour representations may vary, the choice of behaviour representations need to be taken into account when creating the domain component framework. Further, the framework needs to combine the execution of the supported behaviour specifications with the predefined domain behaviour, by, e.g., triggering the execution of the behaviour defined by the application developer at appropriate times in the framework. This also means that the domain component framework may have to define the main flow of control for the runtime environment, although in some cases the issue may be left to the runtime environment itself.

As the domain runtime environment have to be tightly connected with the domain component framework, it may be difficult to make reusable tools or components to support the development beyond what is described for the domain component framework. It is, for instance, probably not worthwhile to try to define a generic runtime environment, as the execution models and as well as the predefined parts of the user interface will vary too much between domain runtime environments. An alternative could be to define one or more “skeletal applications” with basic facilities like opening an application, storing and retrieving runtime state of domain applications. Such “skeletal applications” could be starting points which could be modified and filled in to create a domain runtime environment.

Although the application developer role is defined to not require professional software development expertise, in some cases the person having this role or other persons collaborating with the application developer may have this expertise. In such cases, it should be possible to specialise the runtime environment further for the application under development, e.g., by modifying the predefined user interface parts provided by the environment. For the role playing game case this would, for example, allow adding a more unique look and feel to the game – which is usually desirable in the gaming industry. A certain degree of modification of the user interfaces may however also be defined as part of the domain component framework, by including entities representing user interface issues in the framework. These entities could then be edited along with other application entities using the domain composition environment.

8.1.4 Tool Support for Creating Extensibility Support

As is described in Section 5.3.1, the use-case *create extensibility support* covers the activity of making sure that the domain component developer receives what is needed to easily create components for the environment. This is also connected to Section 8.2, which describes the tool support for the domain component developer. The necessary support for domain component development could either be packaged with the environment, or could be provided in a separate packaged target for the domain component developer only.

The foundation for the extensibility of the environment is already provided by the domain component framework, along with the Foundation and Editing Framework. In addition to the domain specific environment itself, the environment developer should make available the necessary part of the model of the domain component framework to allow models of domain components to be derived for it. If providing the full model of the domain component framework is not an option (e.g., because the developer wants to keep part of it secret), a tool could be used to create a read-only or stripped down version of the model. Similarly, the Environment Builder could contain a tool allowing a read-only or stripped down version of the environment definition to be created, which would give the domain component developer the necessary support for extending the environment without giving away the full version of the environment definition.

The extensibility support should also include documentation of the domain component framework, assisting the domain component developer in getting a good understanding of how the framework can be extended, but documentation tools are outside the scope of this work.

8.2 Tool Support for the Domain Component Developer

The main use-cases for the domain component developer were described in Section 5.4. The tasks of the domain component developer are in many ways similar to the tasks of the environment developer. However, while the environment developer creates a new domain framework and editing environment, the domain component developer has a more limited focus on building components that fit into the existing framework and environment. The tools proposed for the environment developer, including modelling, configuration and code generation tools, are thus a natural starting point for proposing extended or modified tools with a focus on component development.

The domain framework defines the extensibility, and thereby guides and restricts the kind of components which the domain component developer can extend the framework with. For the *design component* use-case, the domain component developer needs a good understanding of the existing framework and where it can be extended, and to acquire this understanding depends on good documentation and clearly defined extensibility in the framework. Components may include both new entities and new service implementations. The design for the component and how it fits into the domain framework could be supported by the same modelling tools and stereotypes used to create the framework, but in addition, this work also requires a (partial) model of the domain framework which allows the component to be related to the framework. This raises questions of how the necessary details of the domain framework model can be shared without giving away the full model, which may be unacceptable for economical

and copyright reasons, but also to avoid unwanted changes to the framework itself. These questions are considered as future work, and are not addressed here.

For the *implement editing support* use-case, the component developer needs support for developing, configuring and integrating editors for the domain components into the editing environment. This involves reusing or creating editors for the Editing Framework, and will in most cases be similar to the *configure property editor for domain classes* use-case of the domain environment developer. The Form Editor discussed for the environment developer should be a suitable tool also for the component developer. As the domain component environment developer extends an existing environment, the main structure editors and categories of editors have already been defined. The editors for the domain component should be fitted into this environment, and to assist with this task, it would be natural to use the Environment Builder with the existing environment as a starting point. However, the component developer should probably be limited to only extending the environment, and not change other parts of it. One possibility would be to support an “environment extension mode” in the Environment Builder, where an existing environment is imported as a “background” which can not be edited, but which gives the necessary guides for how to set up and fit the editors of new components into it.

For the *implement runtime support* use-case, the main task of the domain component developer is usually to implement the domain component from the design. This is governed by the rules of the Foundation Framework and the domain component framework, which helps to ensure that it fits into the domain runtime environment. If the modelling tools and stereotypes used for the domain framework are also used for designing the component, the Code Generator tool defined for the domain environment developer could also be used to generate part of the code for the component. Domain components may in some case also introduce the use of additional behaviour representation and execution components, although this is probably the exception and not the rule.

The *package component* use-case could be supported by a component packaging tool, which could wrap up the domain component along with its associated editor, editor configuration and the editor association needed for fitting the editor into the environment. The packaging tool could be integrated as part of the Environment Builder.

8.3 Tool Support for the Tool Component Developer

On an overview level, the use-cases of the tool component developer are similar to those for the domain component developer. However, while the focus of the domain component developer is to develop components fitting into a particular domain component framework and environment, the focus for the tool component developer is to provide reusable domain independent component. The central areas of concern and examples of reusable parts which was presented in Chapter 6 give an idea of the kinds of tool component that can be envisioned. As the set of possible tool components is quite broad and with great variation in scope among the components, generic tool support beyond support in applying the Foundation and Editing Framework to building tool components have not been identified. The modelling stereotypes and code generation support discussed in the previous sections could be used also here, but

probably will not give substantial support in tasks such as developing new reusable editor components.

As mentioned in Section 6.9, for some of the areas of concern it would be possible to develop more detailed frameworks. For example, a visual editor framework for developing drawing and configuration style editors could give a good starting point for many visual editors. The Graphical Editing Framework (GEF) for Eclipse[16] is, e.g., a recent example of such a framework. In cases where such frameworks are used, it may be possible to develop modelling stereotypes and code generation tools, which could simplify going from a design to an implementation of the component, but this task is left for future work.

8.4 Implementation Status of Frameworks and Tools

This chapter and the previous chapter have outlined an approach to developing domain oriented application composition environments. The descriptions include an overall architecture, the architecture of two component frameworks, and a set of proposed tools that could support the effort of different developer roles. There is no complete prototype implementation of the proposed frameworks and tools, but this section describes the partial prototyping that has been done. The prototyping was done in two phases: initially centred on the Form Editor with connection from the modelling tools, and later centred on the component frameworks and user interface of the Environment Builder. In both phases, Java was used as the prototyping language and platform.

The prototyping in the first phase took place before the ideas of the Foundation and Editing Framework were developed, and thus the implementation was not based on these. In this phase, a working prototype of the Form Editor proposed in this chapter was developed. The prototype used an XML representation of the domain model as input for the tool, and a script was developed for generating this XML representation from a model developed using Rational Rose. The prototype allowed the definition of form-based editors for the domain entities defined in the imported model.

Further, a working prototype of a predecessor to the Generic Editing Environment was also developed. This tool used the editors defined using the prototype Form Editor as input along with the XML representation of domain model. It presented a tree-based hierarchical editor allowing content to be created and edited based on the structures of the imported domain model and forms defined using the Form Editor. This tool used a generic hierarchical content model instead of interacting with classes implementing the domain framework.

The prototyping in this phase also included a simple Code Generator tool which, based on the XML representation of the domain model, generated simple JavaBeans-style skeletal Java code for the domain classes. As this work was done before the component frameworks were developed and before the modelling stereotypes described earlier in this chapter were defined, the Code Generator did not use the mapping described for these and included only mapping from the defined attributes of the domain classes to properties in the JavaBeans model.

[16] GEF is an Eclipse tool project, hosted at: http://www.eclipse.org/gef/

The second phase of prototyping was done during the development of the component frameworks. During this phase, partial prototypes of some of the classes in the Foundation Framework and the Editing Framework were developed. The purpose of this prototyping was to assist in fleshing out and checking some of the details of the frameworks.

In this phase, a partial prototype of the user interface of the Environment Builder was also developed. The figures of the Environment Builder shown in this chapter are screenshots directly from this user interface prototype. An exception to this is Figure 64, where a screenshot from the Form Editor prototype has been glued together with a screenshot from the Environment Builder in an image editor to show how these could be integrated. The Environment Builder prototype does not include any functionality, and is best regarded as just a sketch of the user interface.

8.5 Summary

This chapter has proposed a set of tools which would support the different developer roles and their tasks as defined in Chapter 5. The tool support aims to simplify the development of environments and components built on the Foundation and Editing Frameworks proposed in the previous chapter. The main focus was on tools for the domain environment developer. A set of UML stereotypes was defined along with a mapping to the Java-based implementation of the Foundation Framework. These allow a model a domain framework to be expressed more precisely, and enable tool support for generating skeletal code of the domain framework classes.

The proposed Environment Builder tool with the integrated Form Editor can simplify the task of developing an application composition environment, by transforming part of the work from a programming to a configuration task. The proposed solution is based on importing the domain model, and should allow easy configuration of editors for the domain classes. The same tools could also be used by a domain component developer, but requires that this developer receives a (limited) version of the domain framework model and the environment definition. The proposed component frameworks and tools have not been fully implemented, but limited prototyping have been done for parts of the tools and frameworks, including a working prototype of the Form Editor.

Chapter 9 Discussion and Conclusion

This thesis has presented a new approach to developing extensible application composition environments for end users. The work in the thesis was initiated based on experience from development of proof-of-concept implementations of such environments in two European research projects (described in Chapter 4). This background has helped to identify a set of user and developer roles, to organise these roles into a value chain for development of such environments, and to analyse and describe the use cases for each role (Chapter 5). The approach combines the research areas of component-based software engineering (Chapter 3) and end-user programming (Chapter 2) to propose an overall architecture and component frameworks for building extensible application composition environments (Chapter 7). Further, tools suitable to support the tasks of the different developer roles were proposed (Chapter 8).

This chapter starts by discussing some aspects of the proposed approach, and proceeds to discuss the relation to some recent research and development within the research field. The research method and validity of the results are then discussed, before conclusions are drawn with respect to the research questions and the research problem. Following this, limitations of the research described in this thesis are discussed, before the chapter is closed with directions for future research.

9.1 Discussion of Proposed Approach

This section discusses some aspects of the proposed approach. How the approach addresses software reuse is first discussed, before the approach to end user programming is discussed. It also includes a discussion on how extensibility and tool integration are handled, and the use of document-based storage. The last part of the subsection discusses how suitable current component models are for building extensible application composition environments.

9.1.1 Software Reuse Approach

The proposed approach addresses both development *for* and *with* reuse, which [Sindre 1995] and [Karlsson 1995] identify as the two fundamental aspects of reuse. In the proposed value chain, all the developer roles except the application developer develop *for* reuse by other roles in the chain. Also, in the value chain all developer roles except

the tool framework architect develops *with* reuse of components and frameworks produced by other roles in the chain. The proposed overall architecture and component frameworks can also be seen as enabling a smooth transition from development *for* reuse to development *with* reuse by defining the rules which development of components must follow to ensure their reusability in development of systems.

[Reenskaug 1996] (p. 29) describe four key characteristics of industrialised software production:

- *"An effective software production facility must be specialised";*
- *"Reuse is the only known way to achieve satisfactory quality and efficiency";*
- *"Work and responsibilities must be divided along a vertical value chain";*
- *"Matching task to personnel capabilities".*

The approach proposed in this thesis can assist in achieving these characteristics when organising industrialised production of application composition environments. The approach has defined a value and described the responsibilities of each involved role. Further, development for and with reuse among the roles have been identified, with the proposed architecture, frameworks and tools providing a technical foundation for reuse. The defined roles and technical foundation also support specialised software production, where a tool component developer, e.g., could focus on development of tool components for a specific end-user programming technique. The description of each role can enable matching of task to personnel capabilities, but concrete matching for a particular organisation is outside the scope of this thesis.

9.1.2 End-User Programming Approach

In this thesis, the term *end-user programming* has been used to mean the activity of adding to or modifying the functionality of software, in a way that can be performed without professional programming training. In the value chain, the end-user programming is performed by the application developer. The architecture and tools proposed in this work are based on a development model where the application developer composes the application by building structures of instances from domain classes. The domain classes are part of a domain framework, which predefines most of the behaviour. The application developer adjusts the behaviour within the limits defined by the domain framework and domain classes, through the composition and configuration of the domain instances.

This development model draws on following principles:

- *Domain orientation.* As [Nardi 1993] has documented, domain oriented tools are preferable for end-user programming. The domain framework and domain classes define concepts from the domain, and allow the application developer to create the application using domain terms.
- *Instance-based programming*. Most of the application development is done by creating, structuring and configuring instances of predefined domain object

types. This approach has a different focus than traditional object-oriented programming, which focuses more on defining the object types. The simple mechanism for user-defined types that is proposed, is not very expressive when compared to classical class-based or prototype-based object orientation, but is more adjusted to the expectation that the application developer do not have any formal programming training. This programming model has also been applied with success, e.g., in the Stagecast environment (presented in Section 2.3.5).

- *Division of labour*. The domain framework and predefined classes encode the structuring of the domain and its main behaviour. Thus, the environment developer takes the responsibility for the issues that requires most professional developer training, like defining the architecture for the domain and selecting and implementing advanced algorithms. The application developer can focus most on the content, and needs only override or customise the behaviour when the default behaviour is not suitable.

- *Extensibility of domain*. While the use of domain-oriented tools has been suggested earlier, this work contributes by showing how to use a domain component framework to achieve an extensible solution, and how to more easily integrate such a framework into a composition environment. The extensibility means the composition environments become more open, as third party developers can create components which allows the application developer to include additional concepts which was part of the original composition environment.

This development model attempts to increase what is practically achievable for an application developer by limiting what is theoretically possible. To illustrate this, the model can be compare with a more general tool such as Visual Basic. As Visual Basic is a general purpose language, a wide range of programs can be built with it compared to a domain oriented tool, but as it lacks the high-level domain concepts, structuring mechanisms and pre-made algorithms of a domain-oriented tool, Visual Basic requires more programming skill to utilise this potential.

When compared to the Agentsheets approach ([Repenning 1993]), the proposed approach is similar in using instance-based programming based on a domain oriented application layer. Also, the division of labour is similar, but while the domain layer is crated in the substrate in the Agentsheets approach, we have instead proposed component frameworks and tools to support development of the domain framework using regular professional software development tools (e.g., Rational Rose for modelling, and Java for coding). For the application developer, this details of how the domain oriented application layer is created is mostly hidden. However, our approach supports extensibility of the domain through components. Also, as our choice of notation is not limited by a single substrate, our approach is more open to different behaviour description techniques.

The instance-based composition model in the proposed approach is similar to the OOram Composition System (OOCS) described in [Reenskaug 1996]. In OOCS an instance structure is built by the user by creating and organising instances starting from a “seed” object, and following directions the composition defined by a schema. An OOCS schema is a kind of decision tree, which directs which types of objects can be attached to a given object, and their cardinality.

An approach similar to OOCS was used in the implementation of the prototype of our Form Editor. In this tool, information about the associations between domain objects, including cardinality, derived from the UML model was used for building the structure and detail editors, allowing editors to be specialised for the domain object types. In hindsight, this is an area where our proposed approach could be improved by extending the Foundation Framework to include more of the meta-information corresponding to the OOCS schema. More specifically, information about the allowed containment of or reference to other domain entities could be associated with the Domain Entity Descriptor described in Section 7.3.8 and shown in Figure 44. This could, e.g., be realised by creating an Association Descriptor as a specialisation of Property Descriptor, and the information could be derived from the UML model. This approach would limit the configuration information associated with the editors to include only issues specific to the editing and presentation abilities provided by the editor.

9.1.3 Extensibility and Tool Integration Approach

While the domain component framework ensures extensibility with new concepts from the domain, the foundation framework and editing framework ensures extensibility with new tools within the composition environment. This allows tools developed by different developers to work together and be combined in a single environment.

End-user programming tools have usually attempted to provide all that is needed for development of domain applications within one tool. The tools have tended to build around one single organising metaphor and visual formalism, e.g., the card stack of Hypercard and the grid of AgentSheets and Stagecast Creator. In one way, the metaphor has been the strength of the tool, being an organising principle that speeds and simplifies the development of applications that it is suitable for. At the same time, building parts of an application that do not fit well into the metaphor can be quite frustrating.

The editing framework proposed in this work will allow tools based on different metaphors and visual formalisms to be combined in a single editing environment. The framework allows domain objects to appear in multiple editors, and thus each different aspects of the behaviour of the objects can be described within a suitable tool. In this way the strengths of each tool and metaphor can be utilised, avoiding the weaknesses of having to force parts that do not fit well into a single metaphor.

9.1.4 Document-Based Development Approach

The architecture, frameworks and tools proposed in this work apply a document-based approach to development and storage. An application developer treats the application under development as a document, and the application is stored in a document file. This is a familiar approach to the application developer from wide-spread applications like word processors. It is also the most commonly used approach in existing end-user programming environments, e.g., Hypercard, Stagecast and spreadsheet.

As already discussed in detail in Section 7.4.5, the support for creating good document-based storage is somewhat limited in the selected platform (Java 2 Standard Edition). The problems are mainly how to handle references across the hierarchical structure

during storage of parts of a document, and how to manage different versions of components. These problems have previously been identified and solved, e.g., in OpenDoc. The lack of support on the Java platform may be due to the fact that Java focused on support for the web and on server side applications, while more advanced document handling primarily are required by desktop applications. With the support added in the latest versions of the Java platform, XML-based document handling is partly filling the gap.

When compared to using databases, documents have some benefits when it comes to ease of use for the application developer. Databases frequently require at least some configuration during installation, although this varies between different products. When working with a document that is containing the application, the developer decides when a set of changes should be made persistent. The flexibility that is usually taken for granted when working with documents, e.g., storing the document to a new file if unsure that the changes are what is wanted, is usually not available when working with a database. Also, when the application data is stored in a file, the developer can easily create copies either for backup and version handling purposes, or to use the copy as a template for another application. In some database systems, e.g., Microsoft Access, a single file can contain the database, and thus can be handled in the same way as a document.

From the technical point of view, there are also some benefits with document-based storage compared to a database. Each component that is added to a system will usually, at least in traditional relation databases, requires that one or more tables are added to the structure of the database to store the data managed by the component. Another problem is how to support different storage format for different versions of a component. There are of course numerous situations when it is preferable, or even required, to use databases instead of documents. Applications that store, retrieve and search in large amounts of data usually require a database structure to achieve acceptable performance, but at least for the application domains that were considered in this work, the amounts of data to be handled seem to be in manageable within documents.

9.1.5 Suitability of Current Component Models

A fundamental decision in this thesis was to base the approach on component-based software reuse technology. A relevant question to raise is: based on the experience from the work, how well is the state-of-the-art technology in this area suited to build extensible application composition environments?

One part of such an evaluation could be to consider how well the platform independent descriptions of the proposed component frameworks map to the state-of-the-art component models, such as Java/JavaBeans and .Net. The mapping to Java/JavaBeans of the Foundation and Editing Frameworks was fairly straight-forward. The challenges that did appear where mostly related to how to utilise existing mechanisms of the platform. In some cases this caused a slight deviation from the original design to better conform to the platform. Although the mapping to Java/JavaBeans may have been simplified by the fact that many of the mechanisms in the frameworks were designed based on experience from case studies where Java was applied, it is not expected that mapping to other state-of-the-art (platforms such as Microsoft .NET) should present any major problems.

Current component models such as Java and .NET includes good support for reflection. This enables flexible mechanisms to be built on top of the platforms which can be used to, e.g., interact with behaviour representation components. To determine whether this support is suitable for development of the full range of components in Chapter 6 would, however, require further study, probably including prototyping of some of the components.

The platforms do not directly provide an editing framework comparable to the one proposed, but as is shown from the mapping to Java/JavaBeans for the proposed Editing Framework, this can be built on top of these platforms. In this area, there may still also be mechanisms and experience to be reused from the OpenDoc technology. OpenDoc supported advanced editing framework features such as visual space negotiations and support for non-rectangular areas. A problem here is, however, to find a suitable balance between possibilities and restrictions, as one of the problems in OpenDoc was that it became complex to build simple things.

The topic of document-based development was discussed in the previous section. In this area, the current component models provide limited support. Also in this area, ideas for improvement can be found in OpenDoc which had rich support including features such as storage of cross-references between objects, document version management, robustness for missing components and storage in multiple formats.

9.2 Relation to Recent Research and Development

This section relates our work to recent research and development. This material has not been part of the background for the main part of the thesis work, but may have impact on future work.

The first subsection discusses the relation to model-based development with a focus on Model Driven Architecture (MDA), and is followed by a subsection on relation to the Eclipse platform. Language Workbenches and Microsoft's Software Factories are then discussed, before the last subsection describes relation to research in End-User Development.

9.2.1 Relation to Model-Based Development

In the approach proposed in this thesis, the model of the domain framework plays an important role in the development of an application composition environment. The model is also used as input to some of the tools, including the Environment Builder, Form Editor and Code Generator.

In recent years, there has been a lot of activity around the Model Driven Architecture (MDA) framework adopted by OMG in 2001. MDA is an approach to using models in software development. The approach assists in using the design more actively in the whole life-cycle of software, and automating more of implementation and integration. The primary goals of MDA are portability, interoperability and reusability through architectural separation of concerns [OMG 2003]. To achieve this, a central principle in MDA is to separate the specification of the operation of a system from how it is realised

on a specific platform. MDA uses different levels of abstraction to describe models of a system:

- a *computation independent model* describes the environment and requirements of the system without concern for how the system is structured or details of processing;
- a *platform independent model* (PIM) describes the model of a system from a platform independent viewpoint;
- a *platform specific model* (PSM) adds detail of how a specific platform is used by the system.

Model transformations are used to transform the PIM to a PSM. Elements in a PIM can be marked to indicate its role in a mapping. Marks can, e.g., be types from a model or stereotypes from a UML profile. Transformations can range from manual to fully automated, and the number of transformation steps supported in an MDA approach can vary. Transformations can also be directly to code without using an intermediate PSM. What constitutes a PIM or PSM can depend on viewpoint: a PSM with respect to choice of CORBA middleware can still be a PIM with respect to operating system, and MDA tools can support several steps of transformations to support this [Brown 2004].

Our proposed approach can be seen as applying a MDA approach focused around the use of the domain model. As the proposed tools where prototyped prior to OMG adaptation of MDA, the tools and their description in Chapter 8 has not been rewritten to use more of the MDA terminology and concepts.

The proposed Code Generator tool can be seen as performing a transformation from a PIM directly to code. The model is also used as input to the Form Editor, which can be seen as giving tool support for a manual model transformation. In our approach the domain model is the primary artefact from which the code is generated. [Brown 2004] calls this kind of approach model-centric, as it does not currently define any support for round-trip engineering where changes done to the code can be brought back into the model.

The leading modelling tools, such as IBM Rational Software Modeller, increasingly now includes support for MDA, including model transformations. Using one of these tools, it would probably be possible to implement the proposed Code Generator tool as a customised or user-defined transformation within the tool. [Weis 2003] observes that a generic transformation is not always suitable, and proposes to use a rule-based transformation using a visual notation which also opens for round-trip engineering. For future work on our code generation approach, this can imply that the Domain Environment Developer should be allowed to customise the transformation.

Model-Integrated Computing[17] (MIC) is another model-based development approach, using meta-modelling and model integrity constrains to define domain-specific modelling environments. In MIC, the meta-models are defined using UML class diagrams and UML's Object Constraint Language (OCL). A goal of MIC stated in [Lédeczi 2001], is to enable rapid and cost-effective development of domain-specific

[17] http://www.isis.vanderbilt.edu/Research/mic.html (accessed 25. Jan 2006)

environments. The Generic Modeling Environment toolset of MIC uses meta-models to set up a domain modelling environment in a similar way as our Generic Editing Environment uses an environment configuration to provide a domain specific application composition environment. Differently from our approach, though, MIC is focusing on formal representation, composition, analysis and manipulation of models, and the primary application area of MIC have been embedded computing.

9.2.2 Relation to Eclipse

In the recent years, a lot of interest and activity has built up around the Eclipse platform. Eclipse [Eclipse 2003] is an open extensible Integrated Development Environment (IDE), and aims to be a universal tool platform. The Eclipse platform is open source and royalty free, but allows complementary products to be built from it and sold. From the initial set of tools that became available for the platform, which were mostly focused around Java-based development, the set of available tools and projects built around Eclipse is growing and broadening in scope. The Eclipse platform is built in Java and can be extended with plug-ins.

Eclipse can in many ways be regarded as a parallel to what this thesis suggested, but originally targeted at supporting professional software developers. Eclipse is interesting for the work presented in this thesis from two perspectives:

- Eclipse as an extensible IDE for the domain environment developer (and component developers) ;
- Eclipse as part of a composition environment for the application developer.

When the tools of the domain environment developer were discussed, the use of IDEs was not included as these were considered part of the regular tool chest of professional developers. Eclipse is, however, particularly interesting as an IDE in this context due to its extensibility and the existence of many Eclipse projects building frameworks and tools for the platform. Among the Eclipse projects of interest, the Eclipse Modelling Framework (EMF), UML2, and Model Driven Development Integration projects would be of particular interest for developing the support for accessing and transforming the domain models to partial code. A full implementation of the proposed Code Generator tool would benefit from using these frameworks. The Code Generator tool, the Environment Builder tool, and the Form Editor could also be developed as plug-ins allowing integration in the Eclipse environment.

Of more fundamental interest to this thesis, is whether the Eclipse environment and frameworks could be utilised more directly as part of composition environments for the application developer role. One possibility would be to see if the Eclipse workbench could be used as a basis for the proposed Generic Editing Environment. Most of the activities around Eclipse are focused on tools for professional software developers. Eclipse is, however, claimed to be a universal tool platform. Central to this issue is to determine the usability of the Eclipse workbench UI for people without professional programming background, and an evaluation of this is suggested for future work.

Building on top of Eclipse could have many benefits. The rich set of associated frameworks and tool based on Eclipse would make it more feasible to implement some

of the proposed tools and reusable tool components. For instance, the available Graphical Editor Framework (GEF) and the Visual Editor (VE) project could be evaluated as basis for developing visual editor components such as the structure editors proposed in Section 6.4. Further, the upcoming Graphical Modelling Framework (GMF) project promises to make a bridge between EMF and GEF, allowing easy generation of visual editors based on domain models. Eclipse also have features for managing and updating components as part of the environment, which could cover the component management area of concern described in Section 6.7. Developers already familiar with Eclipse would also be able to draw on this existing knowledge when developing application composition environments, which could contribute to the interest and acceptance of the proposed tools and frameworks.

When considering how the proposed Foundation and Editing framework would be affected by building an implementation of the proposed frameworks and tools based on Eclipse, the mechanisms provided by the Foundation Framework would still be a required. To reuse any similar mechanisms already present in the Eclipse platform or frameworks, such a work should first create a new mapping from the platform independent model to a realisation based on Eclipse. As Eclipse is also Java-based, the current Java mapping would be a suitable starting point.

For mapping the Editing framework to Eclipse, a central issue is how the editors and views defined by the Editing framework should map to the editors, views and perspectives defined by the Eclipse workbench. A difference here is that the Eclipse workbench is built for organising and editing projects consisting of multiple files of different type, while the Editing Framework is designed for supporting hierarchically organised editors within a single document. This indicates that the Editing framework and Eclipse workbench complement each other, with the editing framework taking care of internal issues which is usually left to each Eclipse editor component. However, further work is required to investigate the details of such integration.

To conclude, the Eclipse platform has emerged to be an interesting platform for future work on the proposed frameworks, tools and components proposed in this work. The use of Eclipse as an extensible IDE for the domain environment developer would give good opportunities for integrating developing some of the proposed tools for this role. Further, if the Eclipse workbench is also suitable for the application developer role, this opens a lot of options which could make a full implementation of the frameworks and tools more feasible.

9.2.3 Relation to Language Workbenches and Microsoft's Software Factories

Microsoft is currently developing a new methodology for industrialising software production, called Software Factories[18]. Somewhat simplified, Microsoft's Software Factories can be regarded as a combination of Microsoft's variant of Model Driven Development with Microsoft's alternatives to UML (for domain modelling) and Eclipse. The motivation for this work is observations such as "*Software development, as currently practiced, is slow, expensive and error prone…"* and "*Only modest gains in productivity have been made over the last decade…*" [Greenfield 2004a]. According to

[18] The term *software factory* has also been in use prior to the current Microsoft initiative, usually for efforts on industrialising and automating software development

[Greenfield 2004b] the reason for this is that the current object-orientated paradigm dominating software development does not solve some major chronic problems falling into the following major categories:

- *monolithic construction* of software has still not been replaced by assembly of components at a commercial significant scale;
- *one-off development* where each product is developed like it was unique;
- the *gratuitous generality* of current software development methods gives more degrees of freedom than needed, and modelling languages lacking precision;
- *process immaturity*, with processes tending towards either excessive formalism or excessive autonomy.

To overcome these problems, [Greenfield 2004b] proposes to combine the following set of critical innovations to create a paradigm shift for software development, building on the strengths of object orientation and addressing some of its weaknesses:

- *systematic reuse* can be achieved through software product lines;
- *development by assembly* can be enabled by architecture-driven development approaches, based on self describing components and platform independent protocols;
- *Model-Driven Development* can be used to raise the abstraction level and automate tasks, by use of textual or graphical Domain Specific Languages (DSL) with well defined grammar and meaning;
- *process frameworks* can be used to scale up agile approaches, using constraint-based scheduling and tools providing active guidance.

According to [Greenfield 2004b], it is manageable for most organisations to identify a set of reusable abstractions for the domain and developing runtime support such as a framework or server. However, so far defining automation of the assembly process through domain specific languages with tool support such as editors, compilers, have been out of reach. Microsoft's current tool development efforts attempt to change this situation, making it easier to create tools for domain specific languages.

[Greenfield 2004b] gives the following definition of a software factory: "*A software factory is a software product line that configures extensible tools, process, and content using a software factory template based on a software factory schema to automate the development and maintenance of an archetypical product by adapting, assembling, and configuring framework-based components.*"

Microsoft is currently working on realisation of tool support for Software Factories, centred on extending their Visual Studio environment. The current community technical preview[19] of the Domain-Specific Language Tools, include a project template configuring the Visual Studio environment as a Domain Specific Language Designer.

[19] May 2005 CPT release for Visual Studio 2005 Beta 2

The environment contains a Domain Model Designer (not UML-based), which is used to define the domain model using classes, relationships and value properties. Designer definitions (currently defined using an XML format) are used to set up the notation for a designer, including how diagrams appear and mapping to the underlying domain model.

[Fowler 2005] introduces the term Language Workbenches for the class of tools of which the Microsoft's Software Factories tools are an example. Other examples are Intentional Software[20] and JetBrain's Meta Programming System[21]. All of these systems are still in an early phase of their development. Language Workbenches tools promises to greatly simplify development of domain specific languages with facilities now only found in the best IDE's. These tools typically work directly on the abstract syntax of the language, opening new possibilities in providing multiple perspectives on the code. [Fowler 2005] concludes that Language Workbenches offer two principal advantages: improved programmer productivity through better tools, and the possibility of forging closer relationship with domain experts allowing these to contribute directly to code base. However, it is still too early to say if the second will succeed, and also how Language Workbenches will shape domain specific languages in the future.

There are both significant similarities and significant differences between our approach and the Language Workbench approaches. In our approach more focus has been on reusable visual formalisms and languages components, but the customisation and integration of the domain model and these components done by the Domain Environment Developer can be seen as creating a domain specific language. Compared in more detail to the approach and tools in the Microsoft's Software Factories, we find that in both approaches a central asset is the domain model. In our case this is built using a standard UML tool using our UML profile, while in Microsoft's approach the Domain Model Designer is used. The domain model is in both approaches used in the further work on editors/notations, and the definitions created using our Form Editor also have a similar role to the designer definitions in Microsoft's approach.

There is, however, a difference in motivation and perspective between our work and the Language Workbench initiatives. As our motivation is to create tools enabling end users to build software applications, we have studied existing end-user programming approaches to determine how we can reuse and integrate the best of these techniques in building new environments. The motivation of the Language Workbench initiatives seem primarily to be to improve productivity for professional programmers, and suggest improvements to professional programming approaches which raise the abstraction level through domain oriented languages. Despite these differences in motivation and perspective, the approaches and technology have much in common, and opens interesting opportunities for future research combining the different perspectives.

9.2.4 Relation to End-User Development research

End-user development (EUD) have received increasing interest in recent years, and this has resulted in projects such as EUD-NET and EUSES, and a special issue on EUD in the Communications of the ACM [ACM 2004].

[20] http://www.intentionalsoftware.com/ (accessed 11. Nov 2005)
[21] http://www.jetbrains.com/mps/ (accessed 11. Nov 2005)

The EUD-NET[22] project was a European Network of Excellence on End User Development running in 2002-2003. The goal of the project was to help the European Commission prepare a research agenda in the field of end user development for the next framework programme, and also to increase contact and exchange of ideas between research centres.

The EUSES[23] project is an ongoing collaboration between researches at five American and an English university with the goal of investigating and developing technology enabling End Users to Shape Effective Software. End-user development includes new risks, such as frequent errors causing low dependability of the resulting software, and this motivates the research question of EUSES: "*Is it possible to bring the benefits of rigorous software engineering methodologies to end users?*"

The EUD-NET research agenda and roadmap [Paternò 2003] gives the following definition of EUD: "*End User Development is a set of methods, techniques, and tools that allow users of software systems, who are acting as non-professional software developers, at some point to create or modify a software artefact*". EUD extends the scope somewhat compared to end-user programming, and has more focus on aspects of development such as debugging, testing, and maintenance. This is also reflected in recent research, by an increasing focus on how to adapt and apply software engineering techniques to EUD.

Our work contributes to the EUD research lines described in [Paternò 2003] as follows:

- *Methods, environments, and tools*. Our approach contributes by defining an overall architecture, frameworks, and tools for creating EUD environments. As a part of this we describe how MDA techniques can be used in the development.

- *Architectural issues*. We contribute by defining an overall architecture and component frameworks for extensible EUD environments, allowing different editors and tools to be integrated in a common environment.

- *Programming paradigms and languages* and *interaction techniques*. Although we do not contribute directly to these, our proposed architecture and framework supports integration of multiple languages and interaction techniques.

- *Application domain.* We do not directly address a specific application domain, but describe how a domain model can be used through an MDA-based approach to generate part of the domain framework and to define the editing environment.

- *Organizational factor*. We contribute by defining developer and user roles organised in a value-chain, but do not currently further address organisational issues of end-user development.

[22] http://giove.cnuce.cnr.it/eud-net.htm (accessed 03. Nov 2005)
[23] http://eusesconsortium.org/ (accessed 03. Nov 2005)

Meta-Design and Domain-Oriented Design Environments

[Fischer 2004] proposes an approach to EUD based on meta-design. The objective of meta-design is to engage users in continuous development instead of use of existing systems. In this approach, users are motivated to learn by examples and demonstrations of working systems. Meta-design is an evolution of domain-oriented design environments (DODE) [Fischer 1994]. A DODE combines a construction kit for the domain with tools such as a catalogue explorer for finding design examples, a construction analyser for critiquing user design choices, and an argument illustrator for finding good design examples. An early DODE example is JANUS, which allows a kitchen designer to construct a floor plan, and which provides a catalogue of existing designs to start from, as well as rules for good design triggering critique when violated. A DODE can favourably also be extended to a programmable design environment, by including and end-user programming language [Eisenberg 1994]. The *seeding, evolutionary growth,* and *reseeding* process model is central to support meta-design [Fischer 2004]. In this model, developers and users in collaboration creates an initial seed for the system, which then goes through a phase of unplanned evolution through its use by the users, before developers and users collaborates to do a planned restructuring and enhancement of the system (reseeding). Systems should thus be "underdesigned", allowing users to create own solutions to problems at use time, and system designers should regard their own activity as meta-design, supporting users in being designers of the systems they use.

In our terminology, a DODE is a domain specific application composition environment, and our approach should be well suited to build DODEs. The domain of the DODE would in our approach be described in the domain model, and supported at runtime by the domain framework. In our overall architecture, design catalogues, construction analysers and argument illustrators would be tool components, or if closely tied with a visual formalism they could also be part of the formalism's editor (although this reduces reusability). In our approach, the primary focus has been on the initial development of an application, and not on its evolution during use. Despite this, the *seeding, evolutionary growth,* and *reseeding* process model could also be applied to development in our approach, with the Application Developer (which can also be an End User) and the Domain Environment Developer as prime contributors in the seeding and reseeding. In our approach the Domain Component Developer can also contribute to the evolution of the environment by introducing new components, and this could occur during the evolutionary growth phase, extending the limits of the environment also before the next reseeding.

Component-based EUD and Tailoring

[Mørch 2004] presents a component-based approach to EUD, positioning an end-user developer role between the application assemblers and end-users roles in component-based software developments. Design-environments such as visual builders in IDEs are widely accepted, but are built for professional developers and are not suitable for EUD. To bridge the gap between use and programming of an application and provide a "gentle slope" of learning, [Mørch 2004] proposes to use direct activation and levels of tailoring. Direct activation means that the tailoring functionality should be easily accessible from the use context, e.g., by pressing a modifier key when performing the action that would normally activate the function to be tailored. Three levels of tailoring

are defined: *customisation*, *integration* and *extension*. In a component-based setting, *customisation* means modification of parameters of existing components, *integration* is done by creating or modifying assemblies of components, while *extension* means creating new components (see also the Section 2.4 description of [Mørch 1997] and [Stiemerling 1998]).

Integration is supported by the FlexiBeans component model and the FreEvolve platform. The FlexiBeans component model is based on connections between named ports of the components [Won 2005]. To support recomposition through tailoring at runtime, the FreEvolve platform defines the CAT (Component Architecture for Tailoring) language which is used to describe the composition of an application from atomic components, and which allows hierarchical nesting. A visual component integration formalism has been defined to visualise ports and connect components, but to really support EUD, the ports of components must be defined with semantics facilitating both the user's understanding of how components interact and the computational interaction between components [Mørch 2004]. Findings from a study show that this integration approach works well as long as there is a high ratio of visual to non-visual components.

[Wulf 2005] describes how the FlexiBeans and FreEvolve platform was used in the POLITEAM project to build a customisable search tool for files in shared workspaces. In this project, a search tool was developed in three iterations, and tested in workshops and in real use in parts of the German government. Some of the conclusions drawn from their experiences were that decompositions must be meaningful for the users, and that application whose control flow can be presented as mostly linear are well suited for component-based tailoring. Also, a need for further investigation on whether a single interaction paradigm is suitable for component-based tailorability was noted.

Tailorability has attention towards actual use of the system, and tailoring is performed after initial system design [Won 2005]. This is a difference from our approach, which has focus more on creating new applications. In our approach, *customisation* can be done through detail editors, while *integration* can be done using structure editors by the Application Developer role. *Extension* maps to creating new domain component, which is the responsibility of the Domain Component Developer role in our approach.

The port concept of FlexiBeans is not far from the port concept used in the configuration editor of ComLink (see Section 4.1.1), and is also quite similar to ports in UML 2.0. A port model is not included in our current Foundation Framework, as we originally considered the port concept to be primarily connected to a single visual formalism (configuration editor). Also, there is a partial overlap between ports and the use of named properties containing cross-references in our work. The addition of a port model to the Foundation Framework (along with revision of the property model) is a possible direction for future work, especially if we decide to evolve our frameworks in a direction to support runtime reconfiguration of applications. The FreeEvolve platform stores the description of assemblies of components in CAT files. In our approach, this is part of what should be handled by the persistence mechanism of the Foundation Framework. Regarding the issue of the suitability of a single interaction paradigm for component based tailorability, our frameworks have been designed to support integration of different kinds of editors in an application composition environment.

End-User Developers and Software Engineering

Even though end-user programming is now the most common form of programming, there has been little research into the dependability of the resulting programs [Burnett 2004]. Errors in spreadsheets – the most widely used end-user programming paradigm – are known to be abundant, and in some cases have cost millions of dollars. This illustrates that there is a need for end-user software engineering approaches. However, software engineering tools and techniques used by professional programmers can not simply be repackaged for end-user developers, as the latter group do not have the same background and interest in software development as professional developers. [Burnett 2004] further describes how interactive testing, fault localisation capabilities, and interactive assertions can be introduced in the spreadsheet paradigm, and how motivational devices can be used to spark an end-user programmer's interest in applying these software engineering techniques.

Some recent work has as a first step focused on characterising end-user developers. One example is [Costabile 2003], which discusses the needs of users who are experts in a specific discipline (other than computer science), and analyses the activities these domain expert users perform that may lead to the creation or modification of software artefacts. As designers we are challenged to build environments allowing these users to use the formal language of their domain and tools that resembles those they are familiar with. Another example is [Scaffidi 2005], which describes an ongoing study on how end users represent abstractions, with the goal of uncovering interesting niches of end user programming and uncover key issues for further studies in end user software engineering.

[Ko 2005b] discusses human factors affecting the dependability in end-user development. Among the observations is that the cognitive biases of end-user programmers affect their ability to create correct programs, and in summary the pattern of these biases is that when given the choice, people tend to follow the path of least resistance. End-user programmers often lack a real understanding of the code they create, and this leads to frequent introduction of errors. A possible remedy for the situation, is to create tools allowing the users to focus on the important details, and helping them to minimise the time spent on "unimportant" details. One such tool example is the WhyLine tool [Myers 2004]. In one study, it was uncovered that 50% of all errors were caused by false assumptions in the hypothesis made while correcting existing errors. The WhyLine debugging tool illustrate the runtime behaviour at time of failure based on "why did" questions for unexpected behaviour, and "why didn't" questions for absence of expected behaviour. Results from a study of its use, showed significantly reduced debugging time. In another study, Java programmers using the Eclipse environment for maintenance tasks spend 35% of their time on navigation, and 46% of their time on inspecting task-irrelevant code [Ko 2005a]. Although this study was of professional programmers, it illustrates the potential regarding minimising time used on "unimportant" details. The study further identified three fundamental activities during maintenance work: collect a working set of task-relevant code fragments, navigating dependences within the working set, and repairing or creating code.

Related to our work, results from the works on characterising end-user developers, could be used as input for future refinement of the roles defined in our work and the tasks they perform. Also these works can be used as guidance for the development of application composition environments for end-user programming niches that are

uncovered. In our overall architecture, debugging and testing tools can be included as tool components. An area of future work is to determine to which degree suitable validation tools will be independent of visual formalisms. There may be a conflict here between reusability between visual formalisms, and naturalness and usability for the end-user developer. The techniques suggested for use in the spreadsheet paradigm is for instance somewhat tied to this specific visual formalism, and as such would be simplest to include in an application composition environment as part of the editor for such a visual formalism. One possible approach to extensibility in such cases would be to also develop tool components (e.g. visual formalism editor) as component frameworks, allowing their own tool specific extensions to be added.

9.3 Research Method and Validity of Results

This section returns to the research approach of the thesis described in Section 1.4 and discusses the validity of the results from this thesis. The work presented belongs in the research areas of component-based software and end-user programming, and belongs partly in both the design and experimentation paradigms described in [Denning 2000].

The background for the work is founded on a study of state-of-the-art in research literature and artifacts from end-user programming and software reuse with focus on component-based software. Further, this background is enhanced by practical experience from the two European research projects where proof-of-concept implementations of application composition environments were developed.

The proposed architecture, framework and tools were designed by first analysing the requirements of the different developer and user roles involved, and then applying the knowledge acquired from the theoretical and practical background to the design. As described in Section 8.4, partial prototypes of the frameworks and tools were also implemented, but no full implementation has been made.

The following subsections raise three central questions which are used to discuss the validity of the results and research method of this thesis:

- First, is it possible to implement the proposed frameworks and tools?
- Provided an implementation can be made; can the frameworks and tools be used to create composition environments?
- And, provided they can be used in this way, do they give appropriate support to the developers?

Can the proposed frameworks and tools be fully implemented?

The frameworks and tools proposed in this thesis have only been partly prototyped, so this is a reasonable question to raise. More generally, this also relates to the question of how far an artifact must be prototyped to be acceptable as proof-of-concept. To recall from Section 1.3, the existence of an artifact in the proof-of-concept role proves that a concept is possible to realise, at least in one configuration. The artifact is usually too

complex to derive the behaviour of by using logical reasoning or abstract argument alone.

In the case of the Foundation and Editing Frameworks proposed in this thesis, well-known software patterns were used as a central part of the design. Further, parts of the design were also reused or abstracted from the two case studies in which complete prototypes were built. For parts of the design not directly derived from these sources, and especially in the Editing Framework, prototyping of fragments of the framework were used to check the design. Although no complete implementations of the frameworks have been made, it can be argued that the approach used for its development more or less correspond to a proof-of-concept, and no major obstacles is foreseen for making a complete implementation.

Of the tools, usable prototypes were made of the Form Editor, Generic Editing Environment and the Code Generator components. As was described in Section 8.4 these prototypes were made in an early phase, before the frameworks and UML stereotypes were developed, and thus were not based on these. Also, the Generic Editing Environment was limited to using the forms created with the Form Editor. Still, these prototypes can be regarded as proof-of-concept as they can perform their main tasks as described. What remains to be proved is that this could also be done using the frameworks and UML stereotypes, with the expanded responsibility and generic handling this would require.

For the Environment Builder tool, only user interface sketches were prototyped. Its design maps directly to the concepts of the Editing Framework though, and there should not be any major obstacles in implementing the proposed design based on an implementation of the Editing Framework.

All in all, although full proof-of-concepts prototypes exist only for some of the tools, no remaining major obstacles have been identified which could prevent a full implementation of the proposed frameworks and tools from being made.

Can the proposed frameworks and tools be used to create extensible application composition environments?

If a full prototype of the frameworks and tools existed, an experiment for determining this could be to create an example composition environment and example components which extended it. A success with such an experiment would prove that the frameworks and tools were useable, at least for creating one extensible composition environment.

As a full implementation has not been made of the frameworks and tools, creating a full example composition environment has not been an option in this work. Using the prototypes of the Form Editor and the Generic Editing Environment, it has, however, been possible to do a more limited experiment. Using the Form Editor prototype, the model of the dialogue part of the role playing game example presented in Section 4.4 was imported, and an editor was created. This is illustrated in Figure 64 in Section 8.1.2. The resulting editor was then run using the Generic Editing Environment, and allowed content defined by dialogue model to be edited and structured, as shown in Figure 66 in Section 8.1.2. Using the prototype of the Code Generator, it is also possible

to generate skeletal Java code based on the model, but limited only to generate the class itself and the JavaBeans style get- and set-methods for the properties.

Lacking a usable prototype of the frameworks, no example has been made using these. The frameworks have, however, been based on the experience from the two case studies where complete prototypes of extensible composition environments were implemented. Although the mechanisms were refined and some new mechanisms added in the proposed frameworks, at least no major obstacles can be seen for reimplementing the environments from the two cases based on the proposed frameworks.

With these arguments, it can be concluded that it is probable that the frameworks and tools can be used for their intended purpose, but we can not conclude with the same certainty as if an example composition environment had been created using a full implementation of the frameworks and tools.

Would the proposed frameworks and tools give appropriate support for developers of extensible application composition environments?

The goal for the proposed frameworks and tool is not only that they should be usable, but that they also should be useful and provide some benefit to developers of extensible application composition environments.

To claim empirical support for the appropriateness of the frameworks and tools would, however, require an empirical study based on complete prototypes of the frameworks and tools, as well as a good library of tool and runtime components. Preferably, such a study should test how well a group of developers were able to produce a set of application composition environments using the proposed frameworks and tools, compared to a reference group of developers given free choice to use any other frameworks and tools. Unfortunately, both making such an empirical study and making a complete software prototype of the frameworks and tools, have been beyond the time and resource constraints for this thesis work, but both are good candidates for future work.

In the absence of an empirical study, we address the appropriateness of the proposed frameworks and tools by analysing how they can contribute to a reduction in the development effort required to design, code, and move from design to code in the development of extensible application composition environments.

- *Design*: When using the overall architecture and frameworks, important mechanisms for realising new application composition environment are already in place, reducing and guiding the effort required to design a new environment. The proposed value-chain and the description of the tasks of each role also gives as a clearer division of labour, and for the design effort this means that each developer role can focus on a more limited set of tasks and a have a smaller part of the design to handle. For instance, when the Domain Environment Developer can reuse editors and other tool components, this means that design for these components is also reused.

- *Code*: The frameworks will include implementation a set of important mechanisms, and as this implementation is reused the coding required from the

developers of application composition environments is reduced. When reusable tool components such as editors and behaviour runtime are available, these can greatly reduce the coding required for creating new application composition. Further, the Form Editor tool reduces the coding of detail editors for the domain objects by changing the task from hand-coding to visual configuration.

- *Move from design to code*: The proposed model-driven development tools have the potential to reduce the effort required from moving from design to code, as part of the code is generated directly from the design model by the Code Generator tool. The proposed Form Editor tool also uses the UML model as input, and can be seen as reducing the distance from design to code, although in this case the direct coding is also replaced by visual configuration.

As the proposed frameworks and tools are designed based on studies in end-user programming and component software along with the experience from the two case studies, this further add to the confidence that the frameworks and tools are appropriate for their intended purposes. To conclude, although further work including an empirical study is required for empirical validation, the analysis provided in this section gives us confidence in the appropriateness of the proposed tools and frameworks for developers of extensible application composition environments.

9.4 Conclusion

This section starts by returning to the research questions presented in the beginning of this thesis. After concluding on each of these, it proceeds to drawing a conclusion on the research problem. The main contributions of the thesis are summarised in the last sub-section.

9.4.1 Concluding on the Research Questions

Which user roles and developer roles are involved?

We have identified a set of user and developer roles by first examining the roles involved in component software, end-user programming, and the two case studies. The roles identified are:

- *Tool Framework Architect*. Develops the overall architecture, component frameworks and tools for building application composition environments.

- *Tool Component Developer*. Develops tool components which fit into the frameworks and tools for building environments.

- *Domain Environment Developer*. Creates domain specific frameworks, composition environments and runtime environments using the tools and components provided by the two roles above.

- *Domain Component Developer*. Extends the software created by the domain environment developer by adding new components fitting into the domain framework and composition environment.

- *Application Developer*. Develops domain specific software using the environment provided by the domain environment developer and additional components from domain component developers.

- *End User*. Uses the application constructed by the application developer.

The identified set of roles and their tasks gives a clear division of labour, and forms a value chain for development and use of domain oriented application composition environments. Organising the development based on the proposed value chain would be a step in industrialising and streamlining the development effort, and the proposed division into roles seek to utilise the speciality of each developer, thereby contributing to reducing the development effort required in building extensible application composition environments for end users.

A realisation of the proposed value chain could be viable both in a commercial and a free open source setting, or a combination of these. In both a full commercial setting and open source setting, an implementation of the frameworks and tools preferably bundled with a basic set of tool components would constitute a suitable starting point allowing both the Domain Environment Developer and Tool Component Developer to develop free or commercial products. For the Tool Component Developers commercial opportunities include development of advanced editor components based on visual formalisms either meant for replacing or coexisting with tool components from of the bundled set of components. Successful composition environments products built by the Domain Environment Developer could also open business opportunities for plug-ins created by the Domain Component Developer.

What kind of reusable parts can be identified from existing end-user programming environments?

We have described areas of concern which existing environments from the end-user programming community and the case studies deal with. Reusable part kinds dealing with the concerns were identified and discussed. The areas of concern include:

- languages, translation, runtime representation and execution of behaviour;

- editing of structure, detail and behaviour;

- debugging and testing;

- component management, packaging and deployment.

In existing environments, some of the reusable part kinds cover multiple areas of concern or have dependencies requiring tight integration with other components.

What kind of software architecture is suitable to build such environments?

We have proposed an overall architecture for building domain specific application composition environments based on a non-strict layer model. Central to the architecture is the use of component frameworks to define the rules behaviour for components and mechanisms supporting the development of the layers above. Resting on a standard

component technology platform, a Foundation Framework defines a set of mechanisms and rules supporting both generic runtime components and the higher level component frameworks. An Editing Framework builds on this foundation to define a set of rules for interoperation for editor components.

The domain environment developer defines a domain specific component framework based on the Foundation Framework and a set of selected runtime components. The domain specific application composition environment is then built by selecting and possibly specialising a set of reusable editor and tool components, and integrating these with the domain framework.

How can reusable parts be integrated in new environments?

We have proposed two component frameworks assisting the integration of reusable components into new environments: the Foundation Framework and the Editing Framework. Component frameworks enable extensibility and reuse by specifying a focused architecture around a set of key mechanisms, and defining the rule of behaviour for components.

The proposed Foundation Framework defines a set of mechanisms, including how to define user defined types and behaviour based on domain objects, how to organise the objects in a hierarchical context, and how repositories for component and instances as well as other services can be accessed through the context. This foundation supports the integration of reusable runtime components and higher level component frameworks.

The proposed Editing Framework defines the rules of behaviour for editor components, enabling reuse of editors and extension of editing environment. It defines a set of mechanisms including how the editors relate to domain objects and user defined types, how a hierarchy of editors is maintained and presented in their containing views, and how to handle the lifecycle of editors and initial views of an environment.

What kind of tools will assist developers in building such environments?

We have proposed a set of tools for assisting the domain environment developer in the construction of application composition environments:

- *UML stereotypes* based on the Foundation Framework which can be used to model the domain framework with more precise semantics;

- *Code Generator tool* for producing partial implementations from a model of the domain framework using the proposed UML stereotypes. An example mapping to a Java/JavaBeans-based implementation was also described;

- *Environment Builder tool* for creating application composition environments by configuring the editors and views based on the domain model as input;

- *Form Editor* component for the Environment Builder, allowing composition of custom detail editors for domain entities using the domain model as input;

- *Generic Editing Environment* using the description produced by the Environment Builder to run a domain editing environment.

9.4.2 Concluding on the Research Problem

The research problem of this thesis was formulated as:

- **How can we streamline and reduce the development effort required to build extensible application composition environments for end users?**

The research problem was further focused by the research questions on which the previous subsection concluded. The approach proposed in the thesis to address the research problem can be summarised in the following points:

- Organise the development as a value chain based on the developer and user roles which can be identified for developing extensible application composition environments. This contributes to streamlining the development effort by giving a clear division of labour.

- Apply a common overall architecture for development of composition environments based on component frameworks. Component frameworks guide the construction of the software, and predefine much of the behaviour. While restricting what is theoretically possible, the frameworks enhance what the developer is able to create in praxis, and provide the foundation for extensibility and integration of components. Through reuse, the predefined frameworks contribute by reducing the effort required to design and code extensible application composition environments. The overall architecture is based on the following component frameworks:

  - a standard foundation framework defining the key mechanisms needed for integration of runtime behaviour and higher level component frameworks, including how to handle user defined types and behaviour;

  - a standard editing framework defining the mechanisms and rules of behaviour allowing a set of editor components to be integrated in a common editing environment;

  - a domain framework created by the domain environment developer, supporting extensions of the environment with new domain components as well as definition of user defined types and behaviour by an application developer.

- Reuse the knowledge and techniques of the end-user programming field through reusable components based on the standard foundation and editing component frameworks. This contributes to reducing the effort required to develop new application composition environments as the reuse of components and frameworks means there less left to design and code.

- Support the different developer roles in the value chain by providing tools based on the component frameworks simplifying and organising their development tasks, including:

  - code generation tool for generating partial code based on a domain model applying UML stereotypes for mapping to the foundation framework, reducing the effort required to go from design to code;

  - configuration tools using the domain model as input and transforming the task of creating a composition environment from a programming to a configuration task, thus reducing the coding required.

9.4.3 Contributions

With suitable domain specific application composition environments, users without professional programmer training are able to create new software. The main contribution from this thesis is an approach to building such environments, and the main points of the approach were summarised in the conclusion to the research problem. The approach combines the research areas of component-based software engineering and end-user programming to propose an overall architecture, component frameworks and tools suitable to support development of such environments. The proposed approach has been further concretised and detailed providing the following contributions:

- a proposed value chain for development of application composition environments, with further analyses of the tasks of each of the identified developer roles;

- an overall architecture for developing extensible application composition environments;

- architecture of a Foundation Framework defining a set of mechanisms and rules of behaviour supporting extensibility and integration of runtime components and higher level component frameworks;

- architecture of an Editing Framework defining a set of mechanisms and rules of behaviour for editor components;

- UML stereotypes for modelling the domain framework, and mapping to a Java/JavaBeans implementation of the Foundation Framework based on the added semantics of the stereotypes;

- an Environment Builder tool allowing the definition of a composition environment by configuring the editors and views to use for editing the structure and content defined by the model of the domain framework;

- a Form Editor component for the Environment Builder supporting composition of detail editors for domain classes based on the model of the domain framework.

9.5 Further Research

This thesis has presented an approach to developing extensible application composition environments for end users. The proposed approach opens a set of possibilities for further research and development, and further work based on the results from this thesis can include, but are not limited to, the following:

- *Mapping to Eclipse*. As discussed in Section 9.2.2, a further evaluation of how the Eclipse platform could be utilised in realising the different parts of the proposed approach is an interesting topic. A central part of such work would be to evaluate how well the Eclipse workbench is suited for use by people without professional programming background, and thus if it could be used directly as part of an extensible application composition environment for end users. Further, Eclipse-based projects such as the Graphical Editor Framework (GEF), the Visual Editor (VE), and Graphical Modelling Framework (GMF) could be evaluated for their suitability to build visual editor components. Regardless of the suitability of the Eclipse workbench for the application developer role, Eclipse can also be evaluated as a platform for building and integrating the tools presented in Chapter 8. Such an evaluation could include evaluation of the Eclipse Modelling Framework (EMF), the UML2, and the Model Driven Development Integration projects as a basis for developing the support for accessing and transforming the domain models to partial code.

- *Full implementation of frameworks and tools*. As part of this thesis work, only partial implementations of the proposed frameworks and tools were made, as described in Section 8.4. Development of a (more) complete version the proposed frameworks and tools is thus a clear candidate for future work, and would open possibilities for extended case studies, empirical evaluation, as well as industrial development of extensible application composition environments. Results from the suggested further work on mapping to Eclipse would be valuable input for decisions on the how to approach the implementation work.

- *Building a component library*. In addition to a full implementation of the proposed frameworks and tools, the environment developer role as defined in this work require a set of tool components to use when developing the composition environment. Examples of such components are found in the reusable parts identified in Chapter 6. Building a library of tool components is to a large degree a development task, but will also give feedback on the suitability of the Foundation and Editing frameworks.

- *Empirical studies*. A (more) complete implementation of the proposed frameworks and tools would open possibilities for empirical studies of actual use of the frameworks and tools, and use of the resulting application composition environments. Studies of interest can range from controlled experiments focusing on selected tasks of a specific developer (e.g. the tool component developer building an editor or behaviour component), to following the development of a complete composition environment, or even setting up projects including the full value chain. Preferably, such empirical studies should be set up as comparative studies with two developer groups: one group using of the

proposed frameworks and tools, and a reference group allowed free choice of any other frameworks and tools.

- *Domain specific case studies*. In addition to general empirical studies of use of the proposed tools and frameworks, another possibility is to perform a study of the proposed frameworks and tools in relation to those already in use in a specific domain. For instance, if the role playing game domain introduced in the example in Chapter 4 was selected, such could start by studying one of the popular game development environments to see how it is used in practice, and to see where the limits are for users with no programming experience. Further, the study could include how the experience from the end user programming community could be applied to improve the environments, and finally how our proposed architecture, frameworks and tools could be applied to realise such an improved environment.

- *Extending the frameworks and tools*. The frameworks and tools proposed in this work could of course be extended in further research. Preferably, such extensions should be developed based on results from empirical studies. A possible direction could be to extend the current instance-based object model and user defined type model to include inheritance or other forms of delegation, and define factories to allow different ways of creating instances (e.g., by normal instantiation, by copy, or by using templates). Another area that could be explored is adjustments needed to the frameworks and tools to support development of systems where document-based storage is not suitable.

- *Industrialising the development*. The thesis has presented some first steps, including definition of the value chain for development. Moving further towards an industrialisation of the development, also requires following up some of the other points in this list. Due to the popularity of the Eclipse platform, building on top of Eclipse could be a strategic move to also attract the necessary “critical mass” of developers.

References

[ACM 2004] ACM. “End-user development: tools that empower users to create their own software solutions”. Communications of the ACM, Vol. 47, Issue 9, September 2004.

[Alexander 1977] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King and Shlomo Angel. “A Pattern Language”. Oxford University Press, New York, 1977.

[Apple 1995] Apple Computer, Inc. “OpenDoc Programmers Guide”. Apple Computer, 1995.

[Bosch 1999] Jan Bosch. “Superimposition: A Component Adaptation Technique”. Information and Software Technology, Vol. 41, Issue 5, pp 257—273, Elsevier B.V., 1999.

[Bosch 2000] Jan Bosch. “Design and use of software architectures. Adopting and evolving a product-line approach”. Addison-Wesley, 2000. ISBN: 0-201-67494-7.

[Brockschmidt 1995] Kraig Brockschmidt. “Inside OLE”. 2nd edition. Microsoft Press, 1995.

[Brown 2004] Alan W. Brown. “Model driven architecture: Principles and practice”. Software and System Modelling, Vol. 3, No. 4, Springer, 2004.

[Burnett 2004] Margaret Burnett, Curtis Cook, and Gregg Rothermel. “End-User Software Engineering”. In [ACM 2004].

[Buschmann 1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. ”Pattern-oriented Software Architecture: A System of Patterns”. John Wiley & Sons, 1996. ISBN: 0-4719-5869-7.

[Cable 1998] Laurence Cable. “Extensible Runtime Containment and Server Protocol for JavaBeans.Version 1.0”. Sun Microsystems, 1998.

[Costabile 2003] Maria F. Costabile, Daniela Fogli, Catherine Letondal, Piero Mussio, Antonio Piccinno. “Domain-Expert Users and their

Needs of Software Development". In EUD-NET D3.4: Proceedings of Session on End-User Development, HCI International 2003.

[Curbow 1995] Dave Curbow, Elizabeth Dykstra-Erickson, Kerry Ortega and Geoff Schuller. "OpenDoc. Human Interface Specification for the Macintosh Implementation". Apple Computer, 1995.

[Curbow 1997] Dave Curbow and Elizabeth Dykstra-Erickson. "Designing the OpenDoc Human Interface". In proceedings of the conference on Designing interactive systems: processes, practices, methods, and techniques. ACM, 1997.

[Cypher 1993] Allen Cypher (editor). "Watch What I Do: Programming by Demonstration". MIT Press, 1993.

[Dahlbom 2002] Bo Dahlbom. "The Idea of An Artificial Science". In B. Dahlbom, S. Beckman & G. B. Nilsson, "Artifacts and Artificial Science". Almqvist & Wiksell International, 2002.

[Demeyer 1997] Serge Demeyer, Theo Dirk Meijler, Oscar Nierstrasz, Patrick Steyaert. "Design Guidelines for 'Tailorable' Frameworks". Communications of the ACM, Vol. 40, Issue 10. ACM, 1997.

[Denning 1989] Peter J. Denning, Douglas E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, Paul R. Young. "Computing as a Discipline", Communications of the ACM, Vol. 32, Issue 1, January 1989.

[Denning 2000] Peter J. Denning. "Computer Science: The Discipline". In A. Ralston, E. D. Reilly, D. Hemmendinger (editors), "Encyclopedia of Computer Science, 4th Ed", John Wiley & Sons, 2000.

[Eclipse 2003] "Eclipse Platform Technical Overview", Object Technology International, Inc., February 2003. Whitepaper available from: http://www.eclipse.org
(accessed 19. Oct 2005)

[Eisenberg 1994] Michael Eisenberg and Gerhard Fischer. "Programmable Design Environments: Integrating End-User Programming with Domain-Oriented Assistance". Human Factors in Computing Systems, CHI'94 Conference Proceedings, pp 431—437, ACM, 1994.

[Fischer 1994] Gerhard Fischer. "Domain-Oriented Design Environments". Automated Software Engineering, 1, 2 (1994), pp 177—203, 1994.

[Fischer 1998] Gerhard Fischer. "Beyond 'Couch Potatoes': Frasom Consumers to Designers". In IEEE (Ed.) 1998 Asia-Pacific Computer and Human Interaction, APCHI'98.

[Fischer 2004] Gerhard Fischer, Elisa Giaccardi, Yunwen Ye, Alistair G. Sutcliffe, Nikolay Mehandjiev. “Meta-Design: A Manifesto for End-user Development”. In [ACM 2004].

[Floch 2006] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjørven. “Using Architecture Models for Runtime Adaptability”. IEEE Software, Special issue on State of the Practice and Future Directions of Software Architecture, Vol. 23, No. 2, 2006.

[Fowler 2005] Martin Fowler. “Language Workbenches: The Killer-App for Domain Specific Languages”. June 12, 2005. http://martinfowler.com/articles/languageWorkbench.html (accessed 19. Oct 2005)

[Gamma 1994] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. “Design Patterns. Elements of Object-Oriented Software”. Addison-Wesley, 1994. ISBN 0-201-63361-2.

[Gosling 2000] James Gosling, Bill Joy, Guy Steele, Gilad Bracha. “The Java Language Specification”. Second Edition, Sun Microsystems, 2000.

[Greenfield 2004a] Jack Greenfield. “The Case for Software Factories”. Microsoft Architect Journal, Journal 3, July 2004.

[Greenfield 2004b] Jack Greenfield, and Keith Short, with Steve Cook and Stuart Kent. “Software Factories. Assembling Applications with Patterns, Models, Frameworks, and Tools”. Wiley, August 2004. ISBN 0471202843.

[Hallsteinsen 2005] Svein Hallsteinsen, Jacqueline Floch, and Erlend Stav. “A Middleware Centric Approach to Building Self-Adapting Systems”. In proceedings of Software Engineering and Middleware 2004, Springer LNCS 3437, pp 107—122, 2005.

[Hamilton 1997] Graham Hamilton (editor). “JavaBeans™. Version 1.01”. Sun Microystems, 1997.

[Harrison 1993] William Harrison and Harold Ossher. “Subject Oriented Programming. (A Critique of Pure Objects)”. In proceedings of OOPSLA ´93. ACM, 1993.

[Hartmanis 1994] Juris Hartmanis. “On Computational Complexity and the Nature of Computer Science”. Turing Award Lecture. Communications of the ACM, Vol. 37, Issue 10, October 1994.

[Head 1998] Peter Head (editor). “Comspec. Final Project Report, Public Version”. TIDE project no. 1169, July 1998.

[IEEE 2000] "IEEE Recommended Practice for Architectural Descriptions of Software-Intensive Systems". IEEE Std 1471-2000. ISBN: 0-7381-2518-0.

[Johnson 1993] Jeff A. Johnson, Bonnie A. Nardi, Craig L. Zarmer, and James R. Miller. "ACE: A new approach to building interactive graphical applications". Communications of the ACM, Vol. 36, Issue 4, 1993.

[Kahn 2001] Ken Kahn. "Generalizing by Removing Detail: How Any Program Can Be Created by Working with Examples". In [Lieberman 2001a].

[Karlsson 1995] Even-André Karlsson (editor). "Software Reuse: A Holistic Approach". John Wiley & Sons, 1995.

[Kay 1977] Allan Kay and Adele Goldberg. "Personal Dynamic Media". Computer 10(3):31-41, March 1977.

[Kay 1993] Alan Kay. "The early history of Smalltalk". ACM SIGPLAN Notices, Vol. 28, No. 3, 1993.

[Kiczales 1997] Gregor Kiczales, et. al. "Aspect Oriented Programming". Proceedings of ECOOP '97. Springer-Verlag, 1997.

[Kiczales 2001] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold. "An Overview of AspectJ". Proceedings of ECOOP 2001, Lecture Notes in Computer Science, Vol. 2072, Springer-Verlag, 2001.

[Kitchenham 2002] Barbara A. Kitchenham, et. al. "Preliminary Guidelines for Empirical Research in Software Engineering". IEEE Transactions on Software Engineering, Vol. 28, No. 8, August 2002.

[Ko 2005a] Andrew J. Ko, Htet Htet Aung, and Brad A. Myers. "Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks". In proceedings of ICSE'05, pp 126—135, ACM, 2005.

[Ko 2005b] Andrew J. Ko, and Brad A. Myers. "Human Factors Affecting Dependability in End-User Programming". In proceedings of the First Workshop on End-User Software Engineering (WEUSE 1). ACM, 2005.

[Krueger 1992] Charles W. Krueger. "Software reuse". ACM Computing Surveys, Vol. 24, Issue 2, pp 131—183. ACM Press, 1992.

[Lédeczi 2001] Ákos Lédeczi, Árpád Bakay, Miklós Maróti, Péter Völgyesi, Greg Nordstrom, Jonathan Sprinkle, Gábor Karsai. "Composing Domain-Specific Design Environments". IEEE Computer, Vol. 34, Issue 11, pp 44—51, 2001.

[Li 2005] Jingyue Li, Reidar Conradi, Odd Petter N. Slyngstad, Christian Bunse, Umair Khan, Marco Torchiano, and Maurizio Morisio: "An Empirical Study on Off-the-Shelf Component Usage in Industrial Projects", Proc. 6th International Conference on Product Focused Software Process Improvement (PROFES'2005), Springer LNCS 3547, pp 54—68, 2005.

[Lieberman 2001a] Henry Lieberman (editor). "Your Wish Is My Command: Programming by Example". Academic Press, 2001.

[Lieberman 2001b] Henry Lieberman, Bonnie A. Nardi and David J. Wright. "Training Agents to Recognize Text by Example". In [Lieberman 2001a].

[Lundälv 1998] Mats Lundälv, Doeko Hekstra, and Erlend Stav. "Comspec, a Java Based Development Environment for Communication Aids". TIDE conference in Helsinki, 1998.

[Maes 1987] Pattie Maes. "Concepts and experiments in computational reflection". In Proceedings of OOPSLA'87, pp 147-155. ACM Press, 1987

[Masui 2001] Toshiyuki Masui. "Composition by Example". In [Lieberman 2001a].

[Masuishi 2001] Tetsuya Masuishi and Nobuo Takahashi. "A Reporting Tool Using Programming by Example for Format Designation". In [Lieberman 2001a].

[McDaniel 1999] Richard G. McDaniel. "Building Whole Applications Using Only Programming-by-Demonstration". PhD thesis. CMU-HCII-99-100. Carnegie Mellon University, 1999.

[McDaniel 2001] Richard McDaniel. "Demonstrating the Hidden Features that Make an Application Work". In [Lieberman 2001a].

[McIlroy 1969] M. Douglas McIlroy. "Mass produced software components". In [Naur 1969], pp 138—155.

[Myers 1998] Brad Myers. "Natural Programming: Project Overview and Proposal". Carnegie Mellon University School of Computer Science Technical Report, no. CMU-CS-98-101 and Human Computer Interaction Institute Technical Report CMU-HCII-98-100. January, 1998.

[Myers 2004] Brad A. Myers, John F. Pane, and Andrew Ko. "Natural Programming Languages and Environments". In [ACM 2004].

[Microsoft 1995] Microsoft Corporation and Digital Equipment Corporation. "The Component Object Model Specification". Microsoft, 1995.

[Mørch 1997] Anders I. Mørch. “Methods and Tools for Tailoring of Object-oriented Applications: An Evolving Artifacts Approach”. PhD thesis. University of Oslo, 1997.

[Mørch 2002] Anders I. Mørch. “Aspect-Oriented Software Components”. In N. Patel (editor), “Adaptive Evolutionary Information Systems”, pp 105—123, Idea Group Publishing, 2002.

[Mørch 2004] Anders I. Mørch, Gunnar Stevens, Markus Won, Markus Klann, Yvonne Dittrich, Volker Wulf. “Component-based technologies for end-user development”. In [ACM 2004].

[Nardi 1993] Bonnie A. Nardi. “A Small Matter of Programming. Perspectives on End User Computing”. The MIT Press, 1993.

[Naur 1969] Peter Naur and Brian Randell (editors), “Software Engineering: Report of a conference sponsored by the NATO Science Committee”, Garmisch, Germany, 7-11 Oct. 1968. Brussels, Scientific Affairs Division, NATO, 1969.

[NRC 1994] National Research Council. “Academic Careers for Experimental Computer Scientists and Engineers”. National Academy Press, 1994.

[OMG 2002] “CORBA Components”, Version 3.0, OMG, June 2002.

[OMG 2003] “MDA Guide”, Version 1.0.1, OMG, June 2003.

[OMG 2004] “Common Object Request Broker Architecture: Core Specification”, Version 3.0.3, OMG, March 2004.

[Ossher 2001] Harold Ossher and Peri Tarr. “Multi-Dimensional Separation of Concerns and the Hyperspace Approach”. In “Software Architectures and Component Technology”, Mehmet Aksit (editor). Kluwer, 2001.

[Pane 2002] John F. Pane, Brad A. Myers, and Leah B. Miller. “Using HCI Techniques to Design a More Usable Programming System”, Proceedings of IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC 2002), pp 198—206, Arlington, VA, September 3-6, 2002.

[Papert 1980] Seymour Papert. “MindStorms: Children, Computers and Powerful Ideas”. Basic Books, 1980.

[Paternò 2003] Fabio Paternò, Markus Klann, Volker Wulf (editors). “Research Agenda and Roadmap for EUD”, EUD-NET Deliverable D1.1+D1.2, IST-2001-37470, 2003.

[Pope 1998] Alan Pope. “The CORBA Reference Guide. Understanding the Common Object Request Broker Architecture”. Addison-Wesley, 1998.

[Preece 1994] Jenny Preece, et. al. “Human-computer interaction”, Addison-Wesley, 1994.

[Reenskaug 1996] Trygve Reenskaug with P. Wold and O.A. Lehne. “Working with Objects. The OOram Software Engineering Method”. Manning Publications Co, 1996.

[Repenning 1993] Alexander Repenning. “Agentsheets: A Tool for Building Domain-Oriented Dynamic, Visual Environments”. University of Colorado at Boulder, Ph.D. dissertation, Dept. of Department of Computer Science,1993.

[Repenning 2001] Alexander Repenning and Corrina Perrone. “Programming by Analogous Examples”. In [Lieberman 2001a].

[Scaffidi 2005] Christopher Scaffidi, Mary Shaw, and Bard Myers. “An Approach for Categorizing End User Programmers to Guide Software Engineering Research”. In proceedings of the First Workshop on End-User Software Engineering (WEUSE 1). ACM, 2005.

[Schmidt 2000] Douglas Schmidt, Michael Stal, Hans Rohnert and Frank Buschmann. “Pattern-oriented Software Architecture: Patterns for Concurrent and Networked Objects”. John Wiley & Sons, 2000. ISBN: 0-471-60695-2.

[Seaman 1999] Carolyn B. Seaman. “Qualitative Methods in Empirical Studies of Software Engineering”. IEEE Transactions on Software Engineering, Vol. 25, No. 4, July/August 1999.

[Shaw 2001] Mary Shaw. “The coming-of-age of software architecture research”. In Proceedings of the 23rd international conference on Software engineering, pp 656—664, IEEE Computer Society, 2001.

[Sindre 1995] Guttorm Sindre, Reidar Conradi, and Even-André Karlsson. “The REBOOT Approach to Software Reuse”, Journal of Systems and Software - special issue on software reuse, Vol. 30, No. 3, pp 201—212, Sept. 1995.

[Smith 1994] David C. Smith, Allen Cypher and Jim Spohrer. “KidSim: Programming Agents Without a Programming Language”. Communications of the ACM, Vol. 37, Issue 7, 1994.

[Smith 2001] David C. Smith, Allen Cypher and Larry Tesler. “Novice Programming Comes of Age”. In [Lieberman 2001a].

[Stav 2000] Erlend Stav. “Component Based Environments for Non-Programmers: Two Case Studies”. In Proceedings of the Fifth International Workshop on Component-Oriented Programming (WCOP 2000), Blekinge Institute of Technology, BTH, Research Report No 15/00, 2000.

[Stiemerling 1998] Oliver Stiemerling and Armin B. Cremers. “Tailorable Component Architectures for CSCW-Systems”. In PDP '98 (6th Euromicro workshop on parallel and distributed processing), pp 302—308, Madrid, Spain, IEEE Press, 1998.

[Sun 2000] Sun Microsystems. “Enterprise JavaBeans™ Specification, Version 2.0”. Sun Microsystems, 2000.

[Sun 2003] Sun Microsystems. “Java™ 2 Platform Enterprise Edition Specification, v1.4”. Sun Microsystems, 2003.

[Svanæs 1993] Dag Svanæs. “Comspec. A Software Architecture for Users with Special Needs”. InterAct’93 and Chi’93 conference companion on Human factors in computing systems. ACM 1993.

[Szyperski 1997] Clemens Szyperski. ”Component Software. Beyond Object-Oriented Programming”. Addison Wesley, 1997.

[TASC 1999] Telematics Applications Supporting Cognition (TASC), “Documentation of application”, Project deliverable D5.2, TIDE DE 3214, 1999.

[TASC 2000] Telematics Applications Supporting Cognition (TASC), “Demonstration report”, Project deliverable D7.2, TIDE DE 3214, 2000.

[Thimbleby 1992] Harold Thimbleby, Steve Jones, and Andrew Cockburn. “HyperCard: an object oriented disappointment”. In “Building Interactive Systems: Architectures and Tools”, edited by Gray, P.D. and Took, R., pp 35—55, Springer-Verlag, 1992.

[Ungar 1991] David Ungar and Randall B. Smith. “SELF: The Power of Simlicity”. LISP and Symbolic Computation: An International Journal, 4, 3, Kluwer Academic Publishers, 1991.

[Vinoski 1997] Steve Vinoski. “CORBA: Integrating Diverse Applications Within Distributed Heterogenous Environments”. IEEE Communications Magazine, Vol 35, No. 2, February 1997.

[Voas 1998] Jeffrey Voas, “COTS: The Economical Choice”. IEEE Software, March 1998.

[Weck 1997] Wolfgang Weck. “Independently Extensible Component Frameworks”. In M. Mühlhäuser (ed.), Special Issues in Object-Oriented Programming, pp 177—183, dpunkt Verlag Heidelberg, ISBN 3-920993-67-5, 1997.

[Wegner 1987] Peter Wegner. “Dimensions of object-based language design”. In Proceedings of OOPSLA-87. ACM, 1987.

[Weis 2003] Torben Weis, Andreas Ulbrich, and Kurt Geihs. “Model Metamophosis”. IEEE Software, Vol. 20, No. 5, 2003.

[Won 2005] Markus Won, Oliver Stiemerling, and Volker Wulf. “Component-based Approaches to Tailorable Systems”. To be published in “End User Development”, editors Henry Lieberman, Fabio Paternó, Volker Wulf. Springer, 2005.

[Wulf 2004] Volker Wulf, and Matthias Jarke. “The Economics of End-User Development”. In [ACM 2004].

[Wulf 2005] Volker Wulf, Helge Kahler, Oliver Stiemerling, and Markus Won. “Tailoring by integration of domain-specific components: the case of a document search tool”. Behaviour & Information Technology, Vol. 24, No. 4, pp 317—333, July 2005.

[Zarmer 1992] Craig L. Zarmer, Bonnie A. Nardi, Jeff Johnson, and James R. Miller. “ACE: Zen and the Art of Application Building”. In Proc. 25th Hawaii Intl. Conf. on System Sciences, number 2, pp 687—698, 1992.